UiO : Department of Informatics
University of Oslo

Optimising Firewall Performance in Dynamic Networks

Ratish Mohan

Master's Thesis Spring 2015

Optimising Firewall Performance in Dynamic Networks

Ratish Mohan

18th May 2015

Abstract

More and more devices connect to the internet, this means that a lot sensitive information will be stored in various networks. In order to secure this information and manage the large amount of inevitable network traffic that these devices create, an optimised firewall is needed.

In order to meet this demand, the thesis proposes two algorithms for solving the problem. The first algorithm will minimise the rule matching time by using a simple condition for performing swapping that both preserves the firewall consistency, the firewall integrity and ensures a greedy reduction of the matching time.

The solution is novel in itself and can be considered as a generalisation of the algorithm proposed by Fulp in the paper 'Optimization of network firewall policies using ordered sets and directed acyclical graphs'.

The second algorithm will read the network traffic and provide network statistics to the first algorithm. The solution is a novel modification of the algorithm by Oommen and Rueda in the paper 'Stochastic learning-based weak estimation of multinomial random variables and its applications to pattern recognition in non-stationary environments'.

It will be shown that both algorithms, through experiments, are able to satisfy the problem of optimising a firewall.

Acknowledgements

First and foremost I would like to thank my supervisor Anis Yazidi for his dedication in helping me write this thesis and for his patience in explaining various relevant topics again and again and again.

I would like to extend a thank you to Boning Feng for discussions regarding the master topic, to Haarek Haugerud for his advice regarding unique names, comments and iptables, even if they seemed like a small contribution, they were vital for my firewall optimiser, and to Kyrre Begnum for his excellent videos on structuring a master thesis.

I extend my gratitude to my fellow students for providing both help and a listening ear when necessary, to the University of Oslo and to the Oslo and Akershus University College for providing excellent facilities for studying and writing my thesis.

And finally I would like to express my sincere gratitude to my family for providing many late night dinners and to all my friends both old and new for providing fun relief from the monotony of writing a thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

As our society becomes more dependant on the interconnectivity offered by the internet as well as the convenience of digital services, so does our need to secure the information stored on these devices and services. Thus, in order to secure the information, computer security becomes an important problem to solve.

However, while computer security is an extensive subject and covers both the security of the physical machines as well as the information stored on them, a more specialised form of computer security is needed, the reason being that more and more devices connect to various networks, chief among them the internet [1]. Thus, network security is the branch concerned with the security in a network; this is a broad field spanning many different concepts such as authentication, access policies, intrusion detection, intrusion prevention and honeypots/honeynets.

A first line of defence in network security is to use a firewall in order to enforce access policies. A firewall, in essence, is a program whose objective is to filter the incoming and outgoing packet traffic on a host or in a network. This is enforced by matching each data packets header information against a predefined set of rules known as the firewall policy. Each rule has an action associated with it, either deny or accept, and this action is what decides whether a packet is dropped or not.

A study of many Internet and private traces shows that the major portion of network traffic matches a small subset of firewall rules, which means that the frequency distribution for some of the traffic properties appears to be highly skewed [2]. Furthermore, when performing packet filtering, each rule in a firewall policy will usually be checked in a sequential order, thus, as the firewall policy increases in size, and combined with a matching rule of a higher order, the firewall filtering overhead will become more costly.

This can easily become a bottleneck in a high speed network under attack or heavy network load [3, 4]. Morover, as the computing power of hosts, the transmission speeds of packets and the complexity of networks continue

to increase; a firewall must correlatively be able to adapt to this change by processing packets at increasingly higher speeds [5, 6]. Because of this, it is desirable to decrease the number of packet matches required in order to reduce a potentially costly filtering overhead as well as the overall packet matching time [2].

Thus, in order reduce the number of required packet matches and ensure that a firewall is able to process packets at an adequate speed, it is crucial to optimise the firewall to have an appropriate rule ordering. This can be achieved by ensuring that the rule ordering is such that the rules most often matched appear at the top of the rules list.

This will reduce the amount of time used to process a packet by reducing the amount of required packet matches, thereby, reducing the packet filtering overhead. Additionally, it will also have the effect of improving throughput of the network as a packet will spend less time being processed.

However, because of intra-rule dependencies, the problem of finding the optimal rule order is NP-hard, thus, one must find a heuristic algorithm to find a sub-optimal rule order.

1.1 Problem Statement

At its core, this thesis aims to find a solution to the problem of optimising the *performance* of a firewall in a *dynamic network*, and in order to do so, it attempts to answer the following questions:

- *How can we optimise the order of the firewall rules in order to cope with dynamic network traffic statistics?*

The term *optimising* in the problem statement is a common concept in the field of computer science. It is generally used to describe improvements done to various facets of the field such as programs or infrastructures. The improvements constitutes ameliorating them to run more efficiently or to improve resource use. Often, a compromise between these requirements is needed in order to reach a predefined optimisation goal based on the available hardware and software.

A *dynamic network* can be described as a network under constant change and activity. Essentially, it is a network where the state of packet traffic is not static, but rather dynamic in that the packet traffic will fluctuate so that one type of traffic won't be dominant for an extended period of time.

The *performance* of a program can be defined as the amount of work executed in relation to the time and resources used. While *network traffic statistics* describes the statistics about the current state of a network such as the number of packets passing through it and their various protocols, ports being used, the latency or the throughput.

Consolidating the above terms in relation to a firewall, the problem statement asks for a solution in which a firewalls performance is optimised by sorting the access control rules according statistics taken from the network such that the firewall is able to adapt to the dynamic nature of the network.

Chapter 2

Background

This chapter outlines the current state of progress regarding firewall optimisation as well as introducing several concepts, technologies and applications used in this thesis.

2.1 Firewalls

According to the Request For Comment, *Benchmarking Terminology for Firewall Performance (RFC 2647)* by David Newman [7], a firewall is defined as "***a device or group of devices that enforces an access control policy between networks***". Firewalls are thus, devices or programs that control the flow of network traffic between networks or hosts [8].

2.1.1 Rules and Packets

In order to understand how a firewall operates it is necessary to understand the relationship between access control rules and the packets they govern. This subsection gives a formal explanation of the terms.

A firewall rule, r, is defined as an n-tuple of ordered fields:

$$r = (r[1], \ldots, r[n]), \text{ for } n > 1$$

The upper bound of n is network specific, however, for an internet firewall it is usually set to five and comprises the following fields: **Protocol**, **Source Address**, **Source Port**, **Destination Address**, **Destination Port** [2, 9, 10]. Each rule has an action field associated with it, the value of the field decides what actions the firewall will take when a match is found. Table 2.1 outlines the values of the action field.

The **Protocol** field specifies a protocol as documented in the IP packet headers protocol field, *Internet Protocol (RFC 791)*. For an internet firewall,

Action	Effect
Accept	Forward the packet
Deny	Drop the packet
Log, Accept	Log and forward the packet
Log, Deny	Log and drop the packet

Table 2.1: The action field values and its effects

this would be either TCP, UDP or ICMP, however, it could also contain a wild card value (*), in which case it will match any protocol [8, 9, 11, 12]. The **Address** and **Port** fields specify the source and destination IP addresses and the source and destination port numbers of incoming and outgoing packets. Both of these fields can be configured to represent a range of values or any value, rather than only one value. Tables 2.3 and 2.2 outlines these types of configurations for the address and port fields respectively.

Notation	Example	Explanation
Wild Card	* or *any*	Port range 0 - 65535
Range	90-94	The given range of port numbers
Single	90	A single given port number

Table 2.2: Notation for the port field in a firewall rule

Notation	Example	Explanation
CIDR	192.0.2.0/24	Address range 192.0.2.0 - 192.0.2.255
Wild Card	192.0.*	Address range 192.0.0.0 - 192.0.255.255
Range	192.0.2.2 - 192.0.2.150	The given range of addresses
Single	192.0.2.2	A single given IP address

Table 2.3: Notation for the address field in a firewall rule

A data packet, p, is defined as an n-tuple of ordered parameters:

$$p = (p[1], \ldots, p[n]), \text{ for } n > 1$$

The upper bound of n is limited by the fields defined in the *Internet Protocol (RFC 791)* [11], however, not all fields in the *IP* header is of importance for a firewall. Thus, a data packet as seen by a firewall is comprised mainly of the following fields [2, 9, 10]:

1. Protocol
2. Source Address IP
3. Source PORT

4. Destination Address IP

5. Destination PORT

These fields correspond to the fields a firewall rule is comprised of.

2.1.2 Firewall Policies

A **firewall security policy** defines how an organisations firewalls should handle inbound and outbound network traffic based on the organisations information security policies. Generally an organisations should conduct risk analysis in order to discover what types of traffic passes through its networks at all times and then how to secure it; a firewall security policy is the implementation of such an analysis [8].

Examples of policy requirements include accepting only necessary IP [11] protocols to pass, authorised source and destination IP addresses, autorised TCP and UDP ports to be used, and certain ICMP types and codes to be used [8].

A formal definition of a firewall policy can be defined thus, let

$$R = \{r_1, r_2, r_3, \ldots, r_N\}, \text{ for } N > 1$$

be the set of ordered firewall rules comprising a policy.

Generally, all inbound and outbound traffic not expressly permitted by the firewall policy should be blocked because such traffic is not needed by the organisation. This practice can also have the benefit of reducing the risk of attacks and decreasing the volume of traffic carried on the organisations networks [8].

Such a firewall policy is considered to be comprehensive if any packet, p, has a match in R. In practice, this is achieved by implementing a ***Default Rule*** [2, 8]. A default rule is a *catch-all* rule. It is usually added to end of a policy and is designed such that it will simply discard any packet not matched in the above rules. Table 2.4 shows an implementation of a comprehensive firewall security policy.

		Source		Destination			
No.	Proto.	IP	PORT	IP	PORT	Action	Prob.
1	UDP	190.1.*	*	*	90	accept	0.0556
2	UDP	190.1.1.*	*	*	90-94	deny	0.0556
3	UDP	190.1.2.*	*	*	*	deny	0.0556
4	UDP	190.1.1.2	*	*	94	accept	0.0556
5	TCP	190.1.*	*	*	90	accept	0.0556
6	TCP	190.1.1.*	*	*	88	deny	0.0556
7	TCP	190.1.1.2	*	*	88-94	deny	0.0556
8	TCP	190.1.2.*	*	*	*	accept	0.0556
9	TCP	*	*	161.120.33.41	25	accept	0.0556
10	TCP	140.192.37.30	*	*	21	deny	0.0556
11	TCP	*	*	161.120.33.*	21	deny	0.0556
12	TCP	140.192.37.*	*	*	21	accept	0.0556
13	TCP	*	*	161.120.33.*	22	accept	0.0556
14	TCP	140.192.37.*	*	*	80	deny	0.0556
15	TCP	*	*	161.120.33.40	80	accept	0.0556
16	TCP	*	*	161.120.33.43	53	accept	0.0556
17	UDP	*	*	161.120.33.43	53	accept	0.0556
18	*	*	*	*	*	deny	0.0556

Table 2.4: A firewall security policy configuration with equal initial probabilities

2.1.3 Packet Matching

In many implementations of firewalls, the rules are stored internally as linked lists [10], thus, a firewall will generally, sequentially compare a packet with a rule. In order for there to be a match between a rule, r_i and a packet, p, the parameters of the packet header must be a subset of all the corresponding fields in the rule.

If,

$$r_i[l], \text{ for } l = 1 \ldots n$$
$$p[l], \text{ for } l = 1 \ldots n$$

represents the ordered fields of the rule r_i and the ordered parameters of the packet header for packet p, then the match between rule r_i and the packet p can be denoted as,

$$p \Rightarrow r_i \iff \forall\, l,\ p[l] \subset r[l], \text{ for } l = 1 \ldots n$$

p matches r_i **if** and **only if** the parameters of p is a subset of the fields of r_i. Because each parameter $p[l_i]$ must match the corresponding field $r_i[l_j]$,

the order of fields in a rule is important to the matching process. Thus, a packet, p can match multiple rules in a firewall R. The matching policy of the firewall decides which rule a packet will match. There are generally three common matching policies used, *Best Match*, *Last Match* and *First Match* [13]

Best Match

A packet is compared against all $r_i \in R$, the rule that matches the closest with the packet is selected and its action consequently executed.

Last Match

A packet is sequentially compared to each rule $r_i \in R$, the last rule that matches, $p \Rightarrow r_i \in R$, is selected and its action consequently executed.

First Match

A packet is sequentially compared to each rule $r_i \in R$, the First rule that matches, $p \Rightarrow r_i \in R$, is selected and its action consequently executed.

Both **Best Match** and potentially **Last Match** will increase the packet matching time, thus, in this thesis, a **First Match** matching policy is assumed.

2.2 Firewall Modelling and Policy Anomalies

This section outlines how to model a firewall and what policy anomalies are.

2.2.1 Rule Intersection

As stated in subsection 2.1.1, the parameter of a rule can contain a range of values, this means that multiple rules can intersect. Two rules, r_i and r_j, intersect if a comparison of their ordered parameters gives a nonempty set; this can be denoted as,

$$r_i \cap r_j \neq \emptyset \iff \forall l,\ r_i[l] \cap r_j[l] \neq \emptyset,\ \text{for } l = 1 \ldots n$$

Table 2.5 shows an example of this.

		Source		Destination		
No.	Proto.	IP	PORT	IP	PORT	Action
1	UDP	190.1.*	*	*	90	accept
2	UDP	190.1.1.*	*	*	90-94	deny
3	TCP	140.192.37.30	*	161.120.33.40	80	deny

Table 2.5: Intersecting and non-intersecting rules

Rule 1 and 2 intersect because rule 2 is a subnet in rule 1 and the port in rule 1 is a subset of the ports in rule 2, in other words, the intersection of rule 1 and 2 yields the the following non-empty set,

$$\{190.1.1.0 - 190.1.1.255,\ 0 - 65535,\ 0.0.0.0 - 255.255.255.255,\ 90\}$$

Rule 3 is completely separate from the other two rules and does not intersect with them. The existence of rule intersections in a firewall policy can limit the amount of valid rule orderings and be the cause of anomalies in the policy. The latter is discussed in subsection 2.2.3.

2.2.2 Precedence Relationships

As described in section 2.1, a firewall policy is defined as an ordered set of firewall rules, R, and each packet, p, will be sequentially compared to a rule, r, like a list. Furthermore, a packet can match multiple rules — this is evident by the different types of matching policies — which means that the order of rules are import and should be preserved. If the order is not preserved when re-ordering the rules and is instead reversed, then a packet might match the wrong rule and violate the policy integrity.

The integrity of a policy is defined as the original intent of the policy. Thus, if R is the original firewall security policy and R' is a reordering of all the rules in R, then in order to maintain the integrity of the firewall policy R in R', a packet, p, must match the same rule and have the same action executed in R' as it would in R.

However, it is important to understand that a firewall is not only comprised of disjoint rules. More often than not, there will be ***Precedence Relationships*** between many of the rules. A precedence relationship is a connection between two or more rules where a rule must appear before another in order for the policy integrity to be kept intact. The reasons that a particular rule should appear before another is further described in the subsection 2.2.3.

In order to accurately model a firewall policy with relationships, a Directed Acyclic Graph, $DAG\ G = (R, E)$, rather than a list, can be used. Here R represents the set of firewall rules in a policy and E represents the set

of precedence relationships between rules. Using a DAG to represent a firewall policy has a couple of advantages over a list representation.

The foremost advantage is that it makes modelling precedence relationships in a firewall easier. Each node in the graph will represent a rule and each directed edge between two nodes will represent a precedence relationship. An edge between rules is determined by finding the intersection between the rules in the firewall, R.

Secondly, the problem of optimising the rule order of a firewall has been shown to be comparable to that of the *single machine job scheduling subject to precedence constraints* problem (see subsection 2.4.1). And because a DAG model can be used to represent the scheduling problem, it would be appropriate to use a similar model in order to a model a firewall policy.

Precedence relationships for table 2.4 are shown in the figure 2.1. The figure displays a directed acyclic graph, DAG, created of the relationships.

2.2.3 Anomalies

As firewall policies expand and introduce new rules or remove old rules, the possibility of anomalies occurring in the policy increases [14–16]. In the paper 'Modeling and management of firewall policies' [15], Ehab S Al-Shaer and Hazem H Hamed outline five types of firewall policy anomalies. Table 2.6 is an example of a firewall policy with anomalies.

		Source		Destination		
No.	Proto.	IP	PORT	IP	PORT	Action
1	TCP	140.192.37.20	*	*	80	deny
2	TCP	140.192.37.*	*	*	80	accept
3	TCP	*	*	161.120.33.40	80	accept
4	TCP	140.192.37.*	*	161.120.33.40	80	deny
5	TCP	140.192.37.30	*	*	21	deny
6	TCP	140.192.37.*	*	*	21	accept
7	TCP	140.192.37.*	*	161.120.33.40	21	accept
8	TCP	*	*	*	*	deny
9	UDP	140.192.37.*	*	161.120.33.40	53	accept
10	UDP	*	*	161.120.33.40	53	accept
11	UDP	140.192.38.*	*	161.120.35.*	*	accept
12	UDP	*	*	*	*	deny

Table 2.6: A firewall security policy with anomalies

Shadowing Anomaly A rule is considered to be shadowed if the rule before it matches all the same packets that this rule matches, thus, the

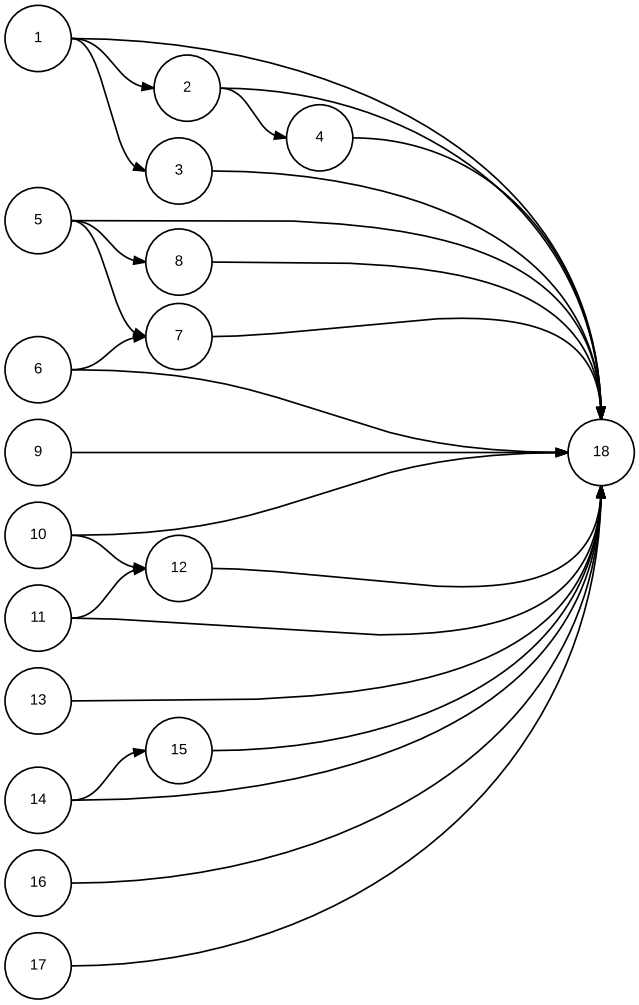

Figure 2.1: Direct Acyclic Graph representation of a firewall security policy

shadowed rule would never be activated. Given two rules, r_i and r_j, rule r_j is considered to be shadowed if the following conditions hold:

- r_j follows rule r_i in the policy rule order
- r_i is a proper superset of r_j

Formally, this can be written as

$$r_i < r_j \ \wedge \ \forall l, \ r_i[l] \supset r_j[l], \ \text{ for } l = 1 \ldots n$$

In the firewall policy given in table 2.6, rule 4 is shadowed by rule 3. In this example, rule 3 accepts traffic where rule 4 denies it, this could be a potentially critical error in the policy as the integrity is seemingly

unsound.

Therefore, according to Al-Shaer and Hamed [15], a general guideline for two rules in an inclusive or exact match relationship, the superset (or general) rule should come after the subset (or specific) rule.

Correlation Anomaly Two rules are considered correlated if their filtering actions are different and both rules filtering parameters are a subset of each other, in other words, that rule r_i matches some packets that matches r_j and vice versa. Formally, this can be denoted as,

$$r_i[action] \neq r_j[action] \ \wedge \ \forall l, \ r_i[l] \subseteq r_j[l] \ \wedge \ r_j[l] \subseteq r_i[l], \ \text{ for } l = 1 \ldots n$$

Rule 1 is in correlation with Rule 3 in the firewall policy from table 2.6. The two rules with this ordering imply that all HTTP traffic that is coming from 140.192.37.20 and going to 161.120.33.40 is denied. However, if their order is reversed, the same traffic will be accepted.

Correlation is considered an anomaly warning because the correlated rules imply an action that is not explicitly stated by the filtering rules. In order to resolve this conflict, Al-Shaer and Hamed [15] write that it is advisable to point out the correlation between the rules and prompt the user to choose the proper order that complies with the security policy requirements.

Generalisation Anomaly A rule is considered a generalisation of a preceding rule if their filtering actions are different and if the first rule can match all the packets that match the second rule. Formally, r_j is considered a generalisation of r_i if,

$$r_i < r_j \ \wedge \ r_i[action] \neq r_j[action] \ \wedge \ r_j[l] \supset r_i[l], \ \text{ for } l = 1 \ldots n$$

Rule 2 is a generalisation of rule 1 in the firewall policy presented in table 2.6. The rules imply that all HTTP traffic that is coming from the address 140.192.37.* will be accepted, except the traffic coming from 140.192.37.20.

Generalisation is often used to exclude a specific part of the traffic from a general filtering action and is considered only an anomaly warning because the specific rule makes an exception of the general rule. This might cause accepted traffic to be blocked or denied traffic to be permitted, Al-Shaer and Hamed [15] advise that it is important to highlight its action to the administrator for confirmation.

Redundancy Anomaly A rule is considered redundant if a preceding rule matches all the same packets and has the same filtering action as the rule such that removing it would not change the integrity of the firewall policy. Formally, given two rules, r_i and r_j, then the rule, r_j, is considered redundant if,

$$r_i < r_j \ \wedge \ r_i[action] = r_j[action] \ \wedge \ \forall l, \ r_i[l] = r_j[l], \ \text{ for } l = 1 \ldots n$$

In the firewall policy given in table 2.6, rule 7 is redundant to rule 6, and rule 9 is redundant to rule 10. Redundancy is considered an error in the firewall policy because a redundant rule adds to the size of the filtering rule list, thus increasing the search time and space requirements of the packet filtering process. In order to avoid redundant rules, Al-Shaer and Hamed [15] advise that a superset rule following a subset rule should have an opposite filtering action.

In some cases redundancy might be preferred to ensure that a desired action is performed on specific traffic, but in general it is considered an anomaly. It is important to discover redundant rules so that the administrator can decide whether to keep these rules, modify their filtering actions, or remove them from the policy.

Irrelevance Anomaly A rule is considered irrelevant if the rule does not match any traffic that can potentially flow through this firewall, that is, when both the source address and the destination address fields of the rule do not match any domain reachable through this firewall. Thus, resulting in the rule having no effect on the filtering outcome of this firewall.

Rule 11 in the firewall policy described in table 2.6, is irrelevant because the traffic that goes between the source (140.192.38.*) and the destination (161.120.35.*) do not pass through this firewall. According to Al-Shaer and Hamed [15], irrelevance is considered an anomaly because it adds unnecessary overhead to the filtering process as it is well understood that keeping the filtering rule table as small as possible helps in improving the overall firewall performance. As such, discovering irrelevant rules is an important function for the network security administrator.

2.3 Taxonomy of Firewall Management Techniques

In the paper 'Traffic-aware dynamic firewall policy management: techniques and applications' [2], Qi Duan and E. Al-Shaer created a tree for classifying firewall policy management techniques in which two top level classifications techniques were defined, namely **Matching Optimisation** and **Early Rejection Optimisation**.

2.3.1 Early Rejection Optimisation

Figure 2.2 outlines the branch detailing the Early rejection optimisation techniques. Such techniques create a minimum set of policy preamble rules (constraints) that can potentially filter out the maximum amount of denied traffic.

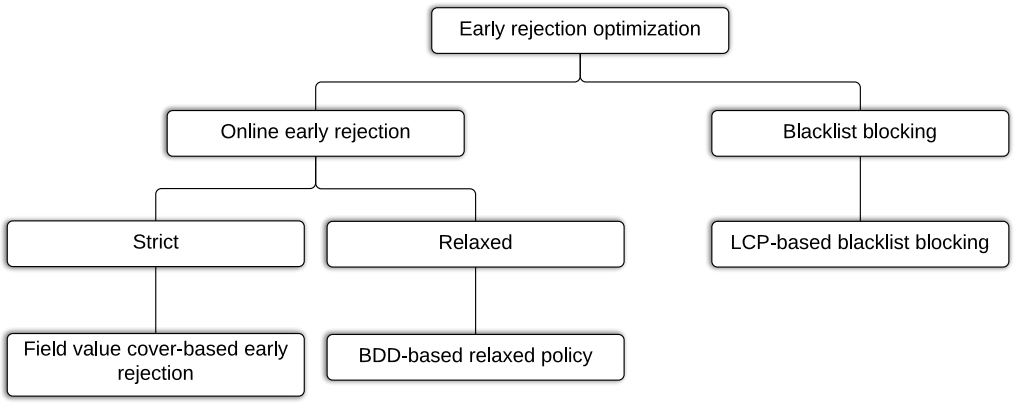

Figure 2.2: Firewall policy management classification tree: Early Rejection Optimisation branch.

According to Duan and Al-Shaer, early rejection optimisation can be classified as either **online early rejection** or **blacklist blocking** [2]. Online early rejection techniques differentiate between the strict *Field Value Cover Early Rejection* policy and the relaxed *Binary Decision Diagram (BDD) based relaxed policy*. Blacklist blocking techniques include the *Longest Common Prefix (LCP) based blacklist blocking* [2].

Online Early Rejection

As stated above, there are two main policies within online early rejection, this section describes them.

Field Value Cover-Based Early Rejection In this technique, an early filtering module is deployed as a layer before the normal firewall policy filtering. The early filtering module attempts to filter out as many rejected packets as possible with the lowest overhead. This reason for this is because dropped packets might traverse a long decision path of rule matching before they are finally rejected by the default deny rule. Such traversal can cause significant overhead proportional to the number of rules in the

policy. The packets on which the early filtering module has made a decision on, either denied or accepted, are not passed on to the original filtering module. If the early filtering module cannot reach a decision based on its approximation of the policy however, the packets are sent to the original filtering algorithm.

The early rejection rules can be created as a combination of the common field values that cover all rules in the policy. Duan and Al-Shaer state that these rules are more practical to find because the number of unique field values are usually few relative to the policy size. Thus, it is desirable to search the firewall security policy for a combination of common field values such that every rule contains at least one of these values. The authors state that this problem can be modelled as a set cover problem. Using a set cover approximation algorithm to can be used to generate a combination of common field values to be used as early rejection rules.

However, finding the correct rules to use in order to achieve an optimal rejection solution is not know in advance. The authors, Duan and Al-Shaer, suggest a suite of three algorithms to address this problem. These algorithms are the *startup phase algorithm*, the *dynamic rule selection algorithm*, and the *early rejection algorithm*. In the startup phase algorithm, the candidate rejection rules list is built from different solutions from the set cover problem. The dynamic rule selection algorithm is responsible for the periodic addition and removal of rules according to each rules performance. The early rejection algorithm calculates the location of early rejection relative to normal packet filtering and defines the per-packet operation of filtering. Duan and Al-Shaer state that the limitation of this technique is that it is not suitable for policies with a large number of rules.

BDD-Based Relaxed Policy The fundamental concept of this technique is to create an approximate policy based on the current policy. This technique then evaluates every packet against the new approximate policy, and decides to either accept or reject the packet. The original policy is deployed as normal, but is not executed unless the early filtering module fails to reach a decision and has to forward the packet to the original policy.

In order to create the new approximate policy, Duan and Al-Shaer state that *Efficient Boolean* expressionss can be used. Thus, in the actual implementation, boolean expressions are used to construct the Binary Decision Diagrams. Each Boolean expression represents the different packets that can match a specific rule, and the variables used for this expression correspond to the bits of individual packet header fields (see section 2.1.1 for valid packet header fields). BDDs can facilitate the matching by representing the expression in the form of a tree, where each variable is checked only once. Because a BDD tree is a type of binary search tree, the decision can be quickly made for a large portion of the packet space.

When a packet arrives, the fields in the packet header are extracted and sorted according to their order in the expression tree, so that they can be used sequentially to traverse the tree. The tree traversal itself is comprised of a set of instructions similar in nature to that of a standard binary search tree. Simply traversing either to the right node or left node depending on the variable at the current node; if true to the left and if false to the right. This is repeated until a node is reached with a final value or the maximum depth allowed in the tree is reached.

According to Duan and Al-Shaer the whole system is dynamic and traffic aware, and can be tuned by the policy structure and previous performance measures. The limitation of this technique is that the overhead to build the BDD is usually significant.

Blacklist Blocking

In Blacklist blocking the fundamental concept is that of blocking specific addresses or ranges of addresses. Longest Common Prefix Based Blacklist Blocking is one such technique.

LCP-Based Blacklist Blocking This technique uses filter selection in order to block addresses. A filter is a set of simple access control rules that specifies which addresses and prefixes should be blocked. The goal of filter selection is to build filtering rules that can minimise the impact of malicious sources in the network using the available network resources. The technique considers different filtering problems based on different attack scenarios, operators' policies and constraints. According to Duan and Al-Shaer, each filtering problem can be modelled as an optimisation problem.

The data structure used to represent the problems is an LCP tree. The LCP tree is a type of binary tree where the leaves of the tree are the malicious IP addresses, and all the other nodes represent the longest common prefixes between any pair of IPs in the tree.

The authors state that the limitation of this technique is that all the malicious IP addresses must be known before the computation of the optimal solution. To defend against attackers who can move quickly among multiple source IPs, one must recompute the optimal solutions frequently. This suggests that this technique may not be efficient enough for a firewall [2].

2.3.2 Matching Optimisation

Figure 2.3 outlines the Matching optimisation branch of the classification tree by Duan and Al-Shaer [2]. This branch is also the focus of this thesis.

Matching optimisation algorithms try to reduce the matching time of normal network traffic, in other words, the objective is to reduce the number of rules to be inspected in the average case. According to Duan and Al-Shaer there are two types of matching optimisation techniques: static and adaptive [2].

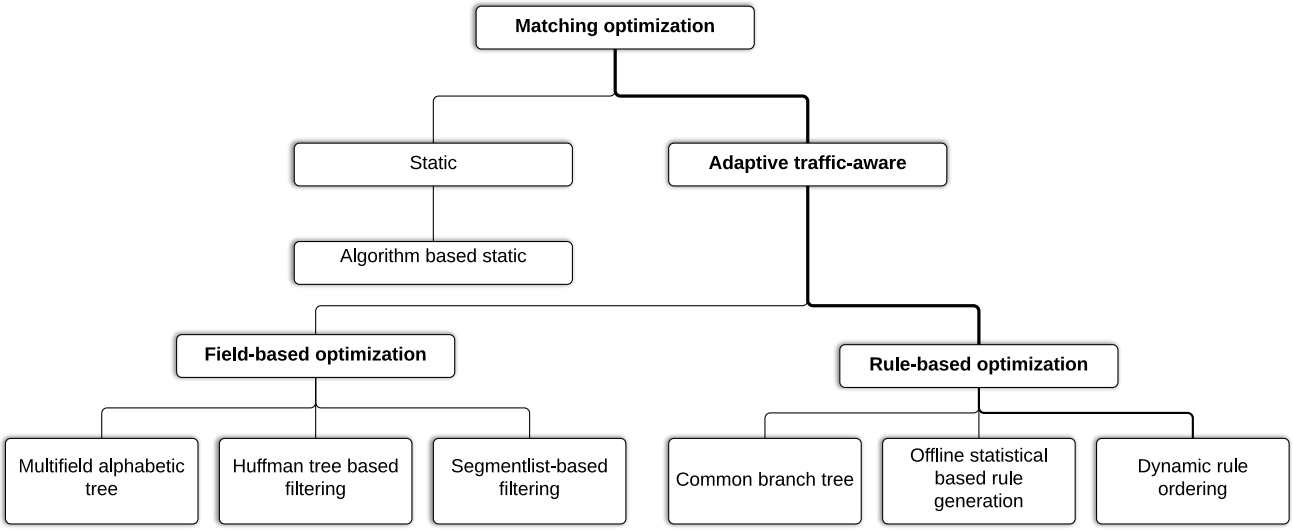

Figure 2.3: Firewall policy management classification tree: Matching Optimisation branch.

Static These filtering optimisation techniques attempt to improve the search time using various algorithmic techniques such as hardware based solutions, specialised data structures, and heuristics. However, these static techniques are used to improve the worst case scenario, and they do not consider the properties of network traffic.

Adaptive traffic-aware These optimisation techniques focus on improving the efficiency of packet filtering in the average case. The techniques can adjust the filtering policies to fit the matching frequency of firewall rules or filtering field values. Duan and Al-Shaer differentiate between two types of traffic aware optimisations, *Rule based optimisation* and *Field based optimisation*.

Rule based optimisation

Rule based techniques include *Common Branch Decision tree*, *Offline Statistical-Based Rule Generation*, and *Dynamic Rule Ordering*.

Common Branch Decision Tree Decision trees can be categorised under three optimisation criteria: worst case, average case, and mixed case. Based on the skewness in the internet traffic, it is suggested that the average case time is an important metric in the packet classification settings and that algorithms based on common branch decision trees can be used to achieve good average case performance [2].

Furthermore experimental evaluation of real-world filters showed that common branching trees use significantly less memory than binary decision trees, while having comparable worst case and average case search times [2]. Duan and Al-Shaer write that the good performance of the common branching tree can be attributed to the presence of extensive wildcarding with a certain structure in the rule sets.

However, the limitation of the technique is that the entire decision tree needs to be rebuilt every time the traffic pattern changes, and it is not appropriate for heavily overlapping rules [2].

Offline Statistical Based Rule Generation An example of an offline statistical based rule generation technique is called *Traffic-aware Firewall Optimiser* (TFO) [2]. The first step in TFO is called pre-optimisation. This step removes all redundancies in the rule set. The second step of TFO uses two optimisers: a ruleset-based optimiser and a traffic-based optimiser. The ruleset-based optimiser contains the *Disjoint Set Creator* (DSC) algorithm and the *Disjoint Set Merger* (DSM) algorithm.

The DSC algorithm converts the original rule set to a semantically equivalent disjoint rule set, which can provide the traffic-based optimiser with full flexibility to re-order rules based on network traffic characteristics. The DSM algorithm merges the rules of the disjoint rule set produced by DSC to reduce rule set size [2]. The traffic-based optimiser has four components: hot caching, total re-ordering, default proxy, and online adaptation.

The hot caching scheme tries to put the rules with the highest matching hit frequency to the top of the firewall policy. The total re-ordering scheme combines the measure of hit frequency and rule size to optimise rule ordering. The default proxy scheme is used to optimise the firewall performance when the default deny rule is being heavily invoked. Finally, the online adaptation scheme builds a long-term rule hit profile to optimise the rule set [2].

Duan and Al-Shaer state that TFO is based on the assumption that only a small portion of rules are dependent on other rules, so it cannot handle

policies with heavily overlapped rules. Furthermore, the rule hit profile is built offline, which limits the adaptability of TFO.

Dynamic Rule Ordering This type of technique uses a heuristic approximation algorithm for optimal dynamic firewall rule ordering based on real-time traffic characteristics [2]. The objective of optimising the firewall rules is to create a semantically equivalent rule order that minimises the packet matching time in firewalls. Duan and Al-Shaer state that the ORO problem is NP-hard, thus, a heuristic approximation algorithm that runs in polynomial time and achieves near-optimal results for the most common firewall policies is needed.

The implementation of the technique also requires a method to compute filtering rule weights in order to capture the matching importance of a given rule relative to others [2]. Each rule in the filtering policy is given a weight that reflects its priority in the packet matching process of the firewall policy. The rule weight is calculated based on the packet matching frequency, which determines how frequent the rule has been triggered, and matching recency, which determines how recent the rule has been triggered during packet matching.

The optimised rule list is constructed based on the computed rule weights and is used for matching upcoming packets to the firewall. Due to the dynamic state of a network connected to the internet, the filtering rules must constantly change to adapt to the current state of the network. The rule weights should thus, be dynamically adjusted to reflect the most recent distribution of network traffic.

According to Duan and Al-Shaer, two types of firewall policy updates are used: performance-based triggered updates and time-based periodic updates. This ensures that an ordering of the policy rules that is as close as possible to the optimal can be computed, while the overhead to compute these updates can be minimised [2]. The limitation of the technique is that it is not good for policies with a large number of overlapping rules.

Field based optimisation

Field value based techniques include *Multi-Field Alphabetic Trees*, *Huffman Tree based Filtering*, and *Segment List Based Filtering*.

Multi-field Alphabetic Tree This type of technique calculates the field value frequency (entropy) and uses this entropy information to build the alphabetic search tree for adaptive packet filtering. The alphabetic search tree stores field values in the leaves based on given weights such that the inherent order of the stored values are preserved. This added constraint of enforcing an order on the placement of values in the tree enables the matching algorithm to branch left or right [2]. Duan and Al-Shaer also state

that the constraint can eliminate the need for preprocessing of the packet field values.

The alphabetic search tree inserts values of higher matching probability and frequency at higher tree levels than the values with less probability. This ensures that the average packet matching time is reduced because there is no need to traverse the whole tree before finding a match on average.The gain in filtering performance is proportional to the degree of skewness in the traffic distribution over field values.

Even in the worst case scenario when the traffic distribution is uniform, this technique cannot do worse than the worst case for binary search trees. The limitation of this technique is that the overhead of updating the tree can be significant [2].

Huffman Tree Based Filtering This technique uses a Huffman tree to represent the distribution of the network traffic addresses in the firewall policy. The Huffman tree can minimise the average number of comparisons applied to packets arriving at the firewall. To build the Huffman tree, the hit count can be used as weights for all addresses in the distribution. The internal nodes of the Huffman tree contain boolean expression in order to match packets. According to Duan and Al-Shaer, this type of technique is good for policies with a large number of rules. The limitation is that the Huffman tree needs to be rebuilt periodically to reflect changes in network flows.

Segment List Based Filtering Is a technique with significantly less maintenance cost than the Huffman tree technique. According to Duan and Al-Shaer, one can use a policy-segment-based search list to utilise the high imbalance in the frequency distribution of packets over the policy segments. One can obtain a very simple yet extremely effective structure by building a simple list of segments that is updated after each packet match. Theoretically, the optimal order is to have the segments sorted in reverse order of their popularity [2]. However, it is impossible to guarantee this without prior knowledge of the exact distribution.

A heuristic algorithm can be used to minimise the average search time. Duan and Al-Shaer state that it is good for extremely biased traffic. The only limitation is that it only works well for a limited amount of time until a good order of segments are obtained.

2.4 Rule Ordering Optimisation Algorithms

As touched upon in section 2.3.2 and seen in figure 2.3, matching optimisation algorithms are one of the foci in this thesis, thus, this section describes relevant rule order optimisation algorithms based on

matching optimisation and outlines the general problem of firewall rule re-ordering.

2.4.1 The Problem of Optimising Rule Re-Ordering

The task of optimising a firewall is comparable to that of the ***Traveling Salesman Problem*** (TSP) [17] with precedence constraints [18]. In, 'On the history of combinatorial optimization (till 1960)' [19] by Alexander Schrijver, the standard TSP is defined as the task of finding the shortest route while traversing each city exactly once, given N cities and their intermediate distances. However, when adding constraints to the problem it becomes more complex.

In the paper 'The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times' [18], the authors, Louis-Philippe Bigras, Michel Gamache and Gilles Savard, examine a variant of TSP with precedence constraints. This variant is know as Time-Dependent Traveling Salesman Problem (TDTSP), in which transition costs between two cities now depends on the time of the visit.

This means that certain cities can only be visited at a given time, thus, trying to find an optimal path with such a constraint means that some cities must be visited before others. In other words, there are dependency relationships between the cites.

This is exactly the problem of finding the optimal rule ordering in a firewall policy with dependency relationships, because finding the optimal rule ordering in a firewall entails creating a rule ordering such that some rules must be "visited" or compared against before other rules, until a match is found.

Finally, the authors, Bigras, Gamache and Savard, state that TDTSP is considered to be a ***single machine job scheduling problem*** [18] and because such a scheduling problem is considered to be NP-hard [20], it follows that the optimisation problem for firewall rules is also NP-hard, thus, lest one does an exhaustive search — which is not a scalable solution — one can only find a sub-optimal solution to the problem using a heuristics based algorithm.

2.4.2 A Simple Rule Sorting Algorithm

In the paper 'Optimization of network firewall policies using ordered sets and directed acyclical graphs' [4], the author Errin W Fulp, designed a simple heuristic algorithm for optimising a firewalls rule ordering as seen in algorithm 2.1.

The algorithm is similar to the bubble sort algorithm in that it compares neighbours and if possible it will swap them. Fulp states that in order

Algorithm 2.1: A simple rule ordering algorithm

Data: A list of firewall rules
Result: A new and improved ordering of firewall rules

```
1 done = False
2 while !done do
3     done = True
4     for (i = 1; i < n; i++) do
5         if (p_i < p_{i+1} AND r_i ∩ r_{i+1} = ∅) then
6             interchange rules and probabilities
7             done = False
```

to preserve the rule precedence relationships, the algorithm uses rule probabilities and rule intersection as the swapping criteria.

For example, suppose there are two rules, rule1 and rule2. Rule1 has a lower probability than rule 2 and the rules don't intersect, this means that the rules are not dependant on each other and are thus swappable. The algorithm will continue until there are no more swappable rules.

However, the problem with this algorithm is that one rule can prevent another from being re-ordered [4], thus, the algorithm is unable to re-order groups of rules. The following is an example of this problem; suppose there are three rules. Rule1, Rule2, and Rule3. Rule 1 and Rule 3 have a dependency relationship. The rules have the following probabilities,

Rule	Prob.
Rule1	0.1
Rule2	0.5
Rule3	0.4

Table 2.7: Small rule example with probabilities

Ideally, the rule with the highest matching probability would appear at the beginning of the rules list in order to reduce the amount of packet matches. Thus, in order to preserve the dependency relationships, the optimal rule order is Rule1, Rule3, Rule2. However the algorithm by Fulp is not able to achieve this rule order the following explains this.

The algorithm will first swap Rule1 with Rule2, it will then check if Rule1 can be swapped with Rule3, but because they intersect, they won't swap. On the second iteration of the While loop the problem is evident; because Rule2 is better than Rule1 they won't swap and because Rule1 and Rule3 intersect they won't swap. Thus when the algorithm is finished, the final order is a suboptimal solution.

However, despite its problems, this algorithm will still create a rule ordering that is better than the original if possible.

2.4.3 Sub-Graph Merging Algorithm

In the paper, 'Towards Optimal Firewall Rule Ordering Utilizing Directed Acyclical Graphs' [12] by A. Tapdiya and E.W. Fulp, the authors present a heuristic algorithm for optimised policy rule re-ordering that is able to re-order a policy containing precedence relationships (or a sub-graph in the **DAG**) in such a way that the policy integrity is maintained.

The algorithm requires certain data structures to work, the following gives a short describes of the most import requirements for the algorithm. The algorithm needs a set, $G(r_i)$, of rules containing the sub-graph rules of r_i, i.e. the dependency relationships for r_i. A FIFO Queue, S, to represent the optimal policy rule sorting; S is initially empty. A list, Q, containing the rules to be sorted; initially this is equal to the original firewall policy, R.

For each pass, the algorithm selects the rule with the highest average subgraph probability from the graph of rules available during that pass. The selected rule is then inserted in to the list of sorted rules, S, if it has no rules dependent on it. Otherwise, the algorithm iteratively sorts the subgraph of its dependents until it finds a rule that has no dependent rules and inserts that rule in the list of sorted rules. The algorithm then updates any data structures and repeats the process until all rules have been placed in S.

2.5 Stochastic Learning Weak Estimation

In 'Stochastic learning-based weak estimation of multinomial random variables and its applications to pattern recognition in non-stationary environments' [21], the authors B John Oommen and Luis Rueda proposed a strategy by which the parameters of a binomial/multinomial distribution can be estimated when the underlying distribution is non-stationary — a non-stationary distribution is a distribution that is susceptible to change over time.

The method, Oommen and Rueda created is referred to as Stochastic Learning Weak Estimation (SLWE), and is based on the principles of the stochastic Learning Automata [22, 23]. The rationale for choosing a weak estimator for non-stationary environments is that the SLWE uses a recursive multiplication-based update form, that achieves the process of unlearning stale data, by an order of magnitude faster than a traditional addition-based updating scheme.

2.6 IPtables

IPtables is program that allows a user to to interact and configure the Linux kernels *Netfilter*. Netfilter is an internal framework for the Linux kernel which is used to manage various network related functions and operations, such as, packet filtering, Network Address Translation (NAT) or port translation.

As mentioned in section 2.1.2, a firewall is considered to be comprehensive if it is able to manage all traffic. This is usually achieved by adding a default rule at the end of a policy. Its purpose is to drop all non-matched traffic, however, in IPtables, this is not a single rule. In order to set a default deny rule in IPtables, one has to set the default policy to drop all packets as seen in Listings 2.1.

```
1 iptables --policy INPUT DROP
2 iptables --policy OUTPUT DROP
3 iptables --policy FORWARD DROP
```

Listing 2.1: "IPtables default rule/policy example"

What this entails is that all other rules defined become exceptions to this default drop policy. In Listings 2.2 an example of a simple IPtables rule is given.

```
1 iptables -A INPUT -i eth0 -p tcp --dport 22 -m state --state NEW -
    j ACCEPT
```

Listing 2.2: "Simple IPtables rule example"

The "-A INPUT" option appends this rule to the INPUT chain, this chain contains all filtering rules for traffic bound to this host. The other two chains are OUTPUT and FORWARD, which handle outbound traffic and traffic destined to another host respectively. "-i eth0" specifies which interface to map this rule to, "-p tcp" specifies which protocol to use, tcp in this case. "–dport 22" denotes which destination port to listen on and because this is an INPUT rule, the destination port refers to port 22 on this machine. "-m state –state NEW" sets the state of the connection to NEW. "-j ACCEPT" sets the rule to accept a packet that matches the fields described above.

Finally, IPtables does not have wildcard symbol such as (*), thus, in order to specify "any" port or addresses, one must simply omit to use the "-s/-d/–sport/–dport" flags in the rules.

2.7 hping3

hping3 [27] is a command-line oriented network tool written by Salvatore Sanfilippo, it is inspired by the linux ***ping*** utility, thus, **hping3** commands follow a similar structure to that of **ping**. Listing 2.3 shows an example of using **hping3**.

```
1 # send 10 TCP packets with a random source address to localhost
2 hping3 --count 10 --rand-source 127.0.0.1
```

Listing 2.3: "hping3 example"

The tool enables its users to primarily send custom TCP/IP packets in order to perform security audits and testing of firewalls and networks. However, it has many other uses as well, the linux manual page for **hping3** [27] outlines some of them:

- Test firewall rules
- Advanced port scanning
- Test net performance using different protocols, packet size, TOS (type of service) and fragmentation
- Path MTU discovery
- . . .

Chapter 3

Approach

This chapter outlines how the thesis proposes to answer the question raised in the problem statement. The approach will describe and explain the necessary steps needed to satisfy it.

3.1 Objectives

The objectives of the thesis pertain to the question raised in the problem statement: **How can we optimise the order of the firewall rules in order to cope with network traffic statistics?** And encompass the following necessary steps that will be described in this chapter,

1. Describing the algorithm for rule re-ordering
2. Describing the algorithm for network traffic awareness
3. Combining the two algorithms in order to create a dynamically optimised traffic-aware firewall
4. Outlining the experiments for the optimisation algorithms
5. Defining the expected results of each experiments

3.2 Finding the Average Matching Time of a Firewall Policy

In the paper, 'An argument for simple embedded ACL optimisation' [28], the authors Vic Grout, John Davies and John McGinn define a metric describing the average matching time of an Access Control List. This metric can be applied to a firewall precisely because a firewall policy is comprised of ACL rules with dependency relationships. The following describes how the metric is calculated.

Let λ_i represent the matching probability of a rule r_i in R, then the average matching time of the rule can be denoted as,

$$r_i * \lambda_i$$

In other words to find the average matching time, simply multiply the rule, r_i's probability with its current position in the firewall. Extending the above to the firewall, R, then the average matching time of the firewall R can be denoted as,

$$\sum_{i=1}^{N} r_i * \lambda_i, \text{ for } N > 1$$

Thus, the average matching time is defined as the amount of rules a packet must be compared against before a match is found. For example, given an average matching time of 2.6 for the policy R this would mean that on average, 2.6 packets will be compared against the rules, r_i, in R before a match is found.

From the above it becomes apparent that to optimise a firewall, the average matching time of the firewall must be low. This can be achieved by ensuring that the rules with high probabilities are at the top of the firewall, because multiplying a large probability number with a low position number will give a smaller number and consequently a lower overall average matching time for the firewall policy.

3.3 The Algorithms

The intent of the problem statement is not to create a rule ordering algorithm alone, but rather, to explore the problem of optimising a firewalls **performance** in a **dynamic network**. This means that for the firewall to have optimised performance at all times, there needs to be a connection between when to re-order the rules as the network traffic begins to favour other rules in the policy.

Thus, two algorithms are needed, one for rule re-ordering and one for updating rule probabilities as the network traffic fluctuates, and finally a program is needed in order to connect the two algorithms into an optimised adaptive firewall.

The rule ordering algorithm should to be able to sort a firewalls rule order based on each rules matching probability, dependency relationships, and firewall position. This will ensure that the average packet matching time is reduced. In order to satisfy these criteria, the algorithm will need to have access to the current firewall security policy and knowledge of each rules dependency relationships and matching probabilities.

The traffic aware algorithm should be able to update a rules matching probability dynamically as the network traffic state changes. This means

that the algorithm will need to have access to the currently applied firewall security policy and the current amount of packet matches for each rule.

Finally, the algorithms must be combined such that they can communicate with each other.The traffic aware algorithm needs to be able to update the probability of a rule and this update must be reflected in the rules used by the rule re-ordering algorithm. Otherwise, the rule re-ordering will never be able to find the optimal rule ordering of a firewall when the network traffic state changes.

3.4 The Experiments

This section describes how the experiments will be conducted. The experiments will be divided into two categories, **Static Experiments** and **Dynamic Experiments**. Static experiments will exclusively test the rule ordering algorithm while the dynamic experiments will test both the rule ordering and weak estimator algorithms.

3.4.1 Environment

The experiments will be conducted on virtual machine instances created on the *Alto Cloud*, cloud service at the *Oslo and Akershus University College of Applied Sciences*. All machines will run on a ubuntu 14.14 server image provided in the cloud.

In order to test the algorithms and the resulting firewalls, two machines are needed. Machine1 (M1) will run the firewall and the optimisation algorithms. Machine2 (M2) will generate network traffic using a traffic generating script. However, because the testing firewalls contain rules with random source and destination IP addresses the traffic generating script can not send the traffic through the internet because it would be lost and never reach the firewall at M1. The reason being that there are no hosts in the environment with those IP address.

Thus, in order to solve this problem, a direct connection between M1 and M2 is needed. This connection will be created by changing M2s default gateway to the IP of M1 so that all traffic from M2 gets routed through M1. This ensures that the spoofed IPs in the network traffic generated by the traffic generating script running on M2 will reach the firewall at M1. Figure 3.1 illustrates the proposed set up.

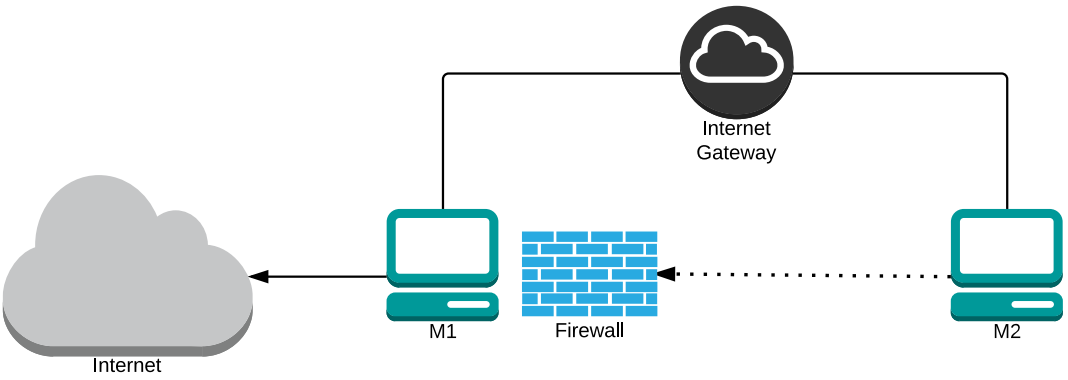

Figure 3.1: Proposed firewall testing environment

The following commands will be used to implement the proposed environment. The first step is to remove the original default gateway so that a new gateway IP can be added instead.

```
# @Machine2
sudo route del gw <existing gateway>
sudo route add gw <Machine1 IP>
```

Listing 3.1: "Changing default route example"

The second step is to enable IPv4 forwarding on the machine running the firewall.

```
# @Machine1
# Turn on IPv4 forwarding so that no packets are dropped
echo "1" | sudo tee /proc/sys/net/ipv4/ip_forward
```

Listing 3.2: "Enabling IP forwarding"

Finally, the below commands will be used to verify that the setup works

```
# Use tcpdump to verify connection between M1 and M2:
sudo tcpdump not port 22 -nnvvS

#Use IPtables to manually check the rules
sudo iptables -xnvL FORWARD
```

Listing 3.3: "Checking the setup"

Once the above has be set up, all traffic from M2 should go through M1 and consequently the firewall, before moving on to the internet.

3.4.2 Firewall Technology

The firewall technology used is the IPtables framework (see section 2.6), thus, the program and scripts will use IPtables commands to interact with the firewall. In order to ensure unique identifiers for each rule; each rule will be written with the commenting option in IPtables. An example of using the comment module is given in Listings 3.4.

```
1 iptables -A FORWARD -i eth0 -p udp -s 190.0.0.0/8 --dport 90 -m
      state --state NEW -j ACCEPT -m comment --comment "A"
2
3 iptables -A FORWARD -i eth0 -p udp -s 190.1.1.0/24 --dport 90:94 -
      m state --state NEW -j DROP -m comment --comment "B"
```

Listing 3.4: "Simple IPtables rules example using the comment module"

In order to test a diverse firewall with both dependent and non-dependent rules, none of the rules are bound for M1, thus, all the rules will be appended to the FORWARD chain.

Naturally, some of the rules are not part of the actual testing, such as the ssh rules, they are there in order to have ssh access to the testing machines in the Alto Cloud.

Finally, each time a new firewall is applied using IPtables rules the rules statistics are reset, essentially, this means that the number of packet matches for a given rule are reset to zero.

3.4.3 Generating Traffic

The traffic generator will be written in python and use the subprocess module to control the hping3 program in order to generate the desired network traffic. The program will generate traffic according to a zipf distribution; for each hping3 command generating traffic to test a firewall rule, it will give them a zipf probability that will determine the hping3 commands chance of being chosen to send data packets to the firewall. The program will use a *roulette wheel* function in order to decide which command to choose based on each commands zipf probability.

3.4.4 Static Experiment 1: Intra-rule re-ordering

The intention of this experiment is to provide proof that the algorithm for optimising the rule ordering is able to reorder the rules such that the rules with the highest probability are at the top of the firewall while still maintaining the integrity of the policy. This experiment specifically tests rule re-ordering within an intra-dependant group of firewall rules.

The experiment will use a small policy of eight rules as described in table 3.1. The rules A - D and E - H are intra-dependent but not inter-dependent.

This means that there are only dependency relationships between the rules group A - D and E - H, but no relationships between the groups themselvs. Figure 3.2 illustrates the relationships in table 3.1 using a *DAG*.

			Source		Destination			
No.	Unique Name	Proto.	IP	PORT	IP	PORT	Action	Prob.
1	A	UDP	190.1.*	*	*	90	accept	0.1147
2	B	UDP	190.1.1.*	*	*	90-94	deny	0.0812
3	C	UDP	190.1.2.*	*	*	*	deny	**0.4286**
4	D	UDP	190.1.1.2	*	*	94	accept	**0.1866**
5	E	TCP	190.1.*	*	*	90	accept	0.0621
6	F	TCP	190.1.1.*	*	*	88	deny	0.0499
7	G	TCP	190.1.1.2	*	*	88-94	deny	0.0415
8	H	TCP	190.1.2.*	*	*	*	accept	0.0353

Table 3.1: A small firewall policy

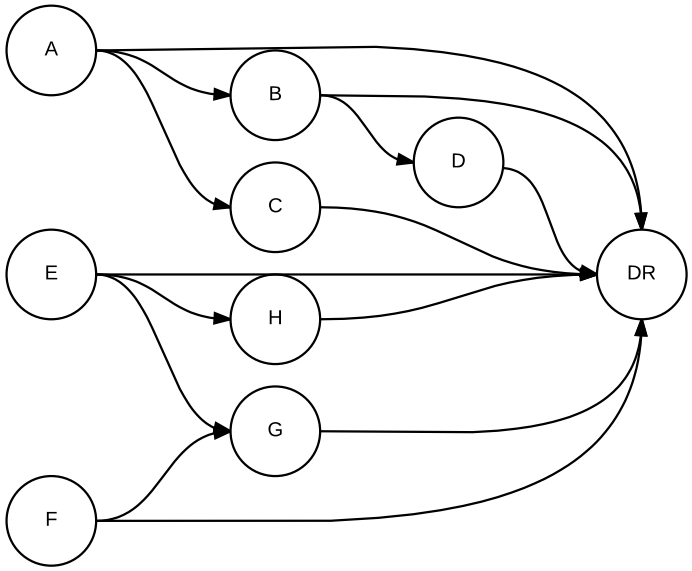

Figure 3.2: DAG of the small firewall policy. DR: Default Rule

From table 3.1 its apparent that this is neither an optimal or sub-optimal rule ordering. The rules C and D have a higher probability than the rules A and B, thus, C and D should be placed higher up in firewall as long as the integrity is maintained. Furthermore this configuration gives the firewall an average matching time of 3.4921.

Expected Results

The expected results from this experiment are as follows,

1. Rule C will be placed higher up than rule B, but below rule A.
2. Rule D, despite having higher probability than rule B, will not be placed higher up in the firewall because of the dependency relationship between rule B and D.
3. The average matching time will decrease when the policy is re-ordered for optimality. It should be 2.6535

3.4.5 Static Experiment 2: Inter-rule re-ordering

The intention of this experiment is to show how the rule ordering algorithm is able to re-order rules with no dependency while still maintaining the integrity of the policy.

The experiment will use a modified version of table 3.1 where the intra-dependent rules, E - H, have a higher probability than that of the rules A - D. Table 3.2 illustrates the new table.

			Source		Destination			
No.	Unique Name	Proto.	IP	PORT	IP	PORT	Action	Prob.
1	A	UDP	190.1.*	*	*	90	accept	0.0621
2	B	UDP	190.1.1.*	*	*	90-94	deny	0.0499
3	C	UDP	190.1.2.*	*	*	*	deny	0.0415
4	D	UDP	190.1.1.2	*	*	94	accept	0.0353
5	E	TCP	190.1.*	*	*	90	accept	**0.4286**
6	F	TCP	190.1.1.*	*	*	88	deny	**0.1866**
7	G	TCP	190.1.1.2	*	*	88-94	deny	**0.1147**
8	H	TCP	190.1.2.*	*	*	*	accept	**0.0812**

Table 3.2: A small firewall policy version 2

As can be observed from table 3.2, the rules E - H should appear at the top of the policy, while rules A - D should be at the bottom. Because the rule groups are independent from each other the policy integrity should be maintained. The average matching time before optimisation for this firewall configuration is 5.1427.

Expected Results

The expected results from this experiment are as follows,

1. The rules E - H will appear at the top of the policy in the same order, while the rule A - D will be at the bottom in the same order.
2. The average matching time will decrease when the policy is re-ordered for optimality. It should be 2.6535

3.4.6 Static Experiment 3: Comparing against Fulps simple algorithm for rule re-ordering

The intention of this experiment is to do a comparison between the rule ordering algorithm from this thesis and the simple rule ordering algorithm by Fulp and to observe the difference in optimality in the resulting policy re-ordering.

In this experiment the actual firewall IPtables script is not important as its only the rule ordering algorithms that are being tested. The only information the algorithms need are the dependency relationships between the rules and each rules matching probability.

The experiment will use a program to generate Directed Acyclic Graphs in order to generate generic dependency relationships and probabilities. The experiment will use DAGs consisting of a 100 nodes (rules). The optimality will be measured by calculating the average matching time of the resulting optimised firewall policies after each algorithm has applied its rule ordering on the policy.

Expected Results

The expected results from this experiment are as follows,

1. Because Fulps algorithm does not take into account dependency relationships between multiple non-neighbouring rules nor each rules position in the policy when deciding to swap, it should generate policies with significantly worse average matching times than the algorithm designed in this thesis.

3.4.7 Dynamic Experiment 1: Schedule based rule re-ordering with dynamic traffic

The intention of this experiment it to test both the rule ordering algorithm and the weak estimator algorithm in a dynamic network using a schedule based re-ordering policy. The schedule policy is based on the amount of packet matches in the firewall.

The experiment will use two zipf distributions based on the firewall in table 3.1. The first distribution, **zipf_dist_X**, will give higher probability to the rules group E - H. The second distribution, **zipf_dist_Y**, will give higher probability to the rules group A - D.

The firewall optimiser script will run the rule re-ordering (RR) algorithm every 100 packet matches generated by the traffic generating script using the zipf_dist_X distribution. After 1000 packets have been matched, the traffic generator will switch the distribution to zipf_dist_Y, while the optimiser script will continue to run RR every 100 packet matches.

For every iteration of the firewall optimiser, the average matching time of the current firewall policy configuration will be calculated using both the current zipf distribution probabilities and the estimated probability values from the weak_estimator algorithm. This will produce the true average matching time and the estimated matching time per packet matched. A base line average matching time will also be calculated, the base line average matching time is simply the the average matching time of the firewall without any rule re-ordering.

Storing these values in tuples, each with the current amount of packet matches at the time of calculation will enable the creation of a graph showing the improvement rate of average matching time for the firewall performance optimiser script.

The graph will consist of two axes, the X axis will represent the total number of packet matches and the Y axis will represent the average matching time. On this graph, the progression of the three different matching time metrics will be plotted.

Expected Results

The expected results from this experiment are as follows,

1. The average matching times will first be very high, before steadily decreasing and once the distribution switch occurs, the matching times should once again sharply increase before decreasing.
2. The exception should be the base line time, which should have a very high matching time until the traffic changes, at which point the average matching time will decrease sharply.
3. The true average matching time should increase and decrease at a sharper rate than the estimated average matching time.

3.4.8 Dynamic Experiment 2: Extended Schedule based rule re-ordering with dynamic traffic

The intention of this experiment is to see the long term effects of the algorithm in a dynamic network using a schedule based re-ordering policy.

This experiment is similar to the previous schedule based experiment, however, while the general setup will be the same, the scale will be different. The experiment will use a larger firewall policy, consisting of 17 rules as defined in Table 2.4 (disregarding the default rule). Consequently, the zipf distributions will also be larger to match the firewall policy.

This time, the traffic generator will start to send data using an initially optimised zipf distribution and after 10000 packets, it will change to a different less optimised distribution.

The rule re-ordering algorithm will run every 1000 packet matches. The average matching times will be calculated in the same manner as before and the values will also be stored in the same manner. The resulting graph will also be similar to the previous experiment, but with higher max values on both axes.

Expected Results

The expected results from this experiment are as follows,

1. The average matching times will first be as low as they can be and should continue to be relatively low and thus optimised, until the switch occurs. Then they should increase at a fast rate, until the rule ordering should cause a steady decreasing again.
2. Again, the base line time should have a low and generally optimised average matching time in the beginning, before sharply increasing once the switch occurs.
3. The true average matching time should increase and decrease at a sharper rate than the estimated average matching time.

3.4.9 Dynamic Experiment 3: Performance triggered rule re-ordering using a sliding window

The intention of this experiment is to observe the performance of the algorithms when using a performance triggered condition. The performance triggered condition will be based on a sliding window comprising the latest values of the estimated average matching time of the firewall.

The experiment will consist of two parts. Both parts will use the same zipf distribution throughout the experiment, however, the first part will shuffle

the zipf distribution at the traffic generator every 500 packets sent. The second part will shuffle the distribution every 1000 packets sent.

The firewall optimiser script will run the rule re-ordering (RR) algorithm according to a performance triggered condition. The condition will consist of a list of the latest average matching times of the firewall. With each new calculation of the average matching time, the value will be added to the list and if the list is full, the oldest element will be removed in order to make space for the latest value. This is the sliding window.

In order to decide whether to run RR or not, the optimiser script will find the trend of the sliding window. If the trend shows that the average matching time is increasing, then RR will be called, else, it will not. The trend will be calculated by finding the average of all but the latest value in the sliding window, the average is then compared against the latest value and if the average is greater then RR will be run, otherwise it wont.

The results will enable us to create a graph, where the X axis will represent the total number of packet matches and the Y axis will represent the average matching time.

Expected Results

The expected results from this experiment are as follows,

1. More rule re-orderings, but the average matching time should stay relatively low throughout the experiment.
2. The greater the sliding window size the better the performance will be, i.e. the average matching time should be higher, the smaller the sliding window is, and lower, the larger the sliding window size is.

Chapter 4

Result and Analysis 1: Implementations

This chapter contains the results of the algorithms, programs and scripts designed and described in the approach section. It also contains proofs for the formulas used in the rule ordering and traffic aware algorithms.

4.1 Traffic Generating Script

In order to test the new firewall configuration created by the performance optimisation algorithms, a network traffic generating script was created as described in the approach. The script itself is a simple wrapper for hping3 written in python and enables the user to configure hping3 commands by passing options and flags to the wrapper script.

The script reads two files containing necessary information. The first file contains the protocol, address, and port information needed to create hping3 commands to test the target firewall. An example is given in Listings 4.1.

```
1 proto,src_ip,src_port,dst_ip,dst_port
2 udp,190.0.0.2,100,*,90
3 udp,190.1.1.10,50,*,92
4 udp,190.1.2.20,55,*,80
5 udp,190.1.1.2,60,*,99
6 tcp,190.1.20.3,85,*,90
7 tcp,190.1.1.8,39,*,88
8 tcp,190.1.1.2,20,*,89
9 tcp,190.1.2.90,51,*,190
```

Listing 4.1: "A small file containing necessary source and destination information for the traffic generating script"

The second file contains a list of numbers from a zipf distribution and represents the probabilities for each hping3 command. An example is given in

Listings 4.2.

```
1 0.42862930043528075
2 0.18657173946957867
3 0.11469285134920415
4 0.08121006644518972
5 0.062132360042377745
6 0.049922963174044155
7 0.04149198453973711
8 0.03534873454458779
```

Listing 4.2: "A small file containing a zipf distribution"

The files have a one to one mapping between each other. This means that the rule created with the information at line 1 in the first file will have the corresponding probability from line 1 in the second file. The probabilities represents each rules likelihood of being chosen by the traffic generator using a roulette wheel function.

Listings 4.3 shows example usage of the traffic generator. As can be seen, the generator takes five arguments, the 'n' flag specifies that hping3 should only output numeric values and not attempt to look up symbolic names for the host addresses.

The 'c' flag specifies a count variable and represent the total amount of packets to be sent, the 'I' flag specifies which interface hping3 should send its packets on, and the 'f' and 'z' flags represent the aforementioned important files.

```
1 sudo python traffic_generator.py -n -c 50 -I eth0 -f
    src_dst_addr_less.csv -z zipf_dist.txt
```

Listing 4.3: "Running the traffic generator"

Finally, a sample output from the script is given below. The first lines in the sample output represent the hping3 command created using the information in the file described by Listings 4.1. The "**- -faster**" option in the command tells hping3 to send packets with an interval of 1 micro second and the "**-S**" option sets the SYN flag in each packet.

The reason for the latter is to speed up the packet sending process, because the generator doesn't need to wait for a reply and that the IPtables firewall will register a lone SYN packet as a packet match for one rule.

Sample output from traffic_generator.py

```
hping3  -n -I eth0 -c 1 --faster --spoof 140.192.37.20 -s 90 -S
        --rand-dest x.x.x.x -p 21

--- x.x.x.x hping statistic ---
1 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

4.2 Rule Ordering Algorithm

The approach section called for a firewall rule re-ordering algorithm which is able to re-order rules based on each rules probability, dependency relationships, and position in the firewall such that the packet matching time is reduced. This section contains the results that satisfies the criteria.

4.2.1 The Algorithm Design

The algorithm uses the simple rule re-ordering algorithm by Fulp (Algorithm 2.1) as a base. The result is shown in Algorithm 4.1.

Algorithm 4.1: A rule re-ordering algorithm

Data: A list of firewall rules
Result: A new and improved ordering of firewall rules

```
1  for rx in rules do
2      Δ_max = 0
3      for ry in rules do
4          Δ_new = 0
5          if rx ≠ ry then
6              if rx.pos < ry.pos then
7                  if rx.pos < succeeding_max(ry) AND
                   ry.pos > preceding_min(rx) then
8                      if rx.prob < ry.prob then
9                          Δ_new = (ry.prob − rx.prob) ∗ (ry.pos − rx.pos)
10                         if Δ_max < Δ_new then
11                             Δ_max = Δ_new
12             else
13                 if ry.pos < succeeding_max(rx) AND
                   rx.pos > preceding_min(ry) then
14                     if ry.prob < rx.prob then
15                         Δ_new = (rx.prob − ry.prob) ∗ (rx.pos − ry.pos)
16                         if Δ_max < Δ_new then
17                             Δ_max = Δ_new
18     if Δ_max > 0 then
19         swap(rx, ry)
```

In order to explain the algorithm, certain important data structures must be described first.

- **The preceding list** of a rule, r_i, contains all the rules that are dependent on r_i, essentially, this means that r_i must appear **before** the rules in the preceding list in order to maintain the policy integrity.
- **The succeeding list** of a rule, r_i, contains all the rules that r_i is dependent on, essentially, this means that r_i must appear **after** the rules in the succeeding list in order to maintain the policy integrity.

The algorithm itself takes as input, a list of rules and has two main loops that iterates through it. For every iteration of the outer loop, the inner loop will traverse the whole list. The reason for this is that the algorithm will compare the current element in the outer loop, rx, with the current element, ry, in the inner loop.

The algorithm will then try to find a **swapping window** between rx and ry. A swapping window is defined as an interval of positions in a firewall in which the two comparing rules can be swapped to without breaking the integrity of the firewall policy. The window is found by looking at the two comparing rule's succeeding and preceding lists.

By finding the rule with the highest position in the firewall in the preceding list for rx and the rule with the lowest position in the firewall in the succeeding list of ry an interval of positions can be found. Once an interval has been found, the algorithm will check if the window is a valid swapping window for the current rules being compared.

In order to check the validity of the swapping window, the algorithm will check if the current position of rx is less than the lowest position in the succeeding list of ry and if the position of ry is grater than the highest position in the preceding list of rx, if the expression is evaluated to *True* then the swapping window is considered valid.

However, the above is only valid if rx has a higher position in the firewall than ry. In the case where ry has a higher position in the firewall than rx (as seen in line 12 and 13 in Algorithm 4.1) there is a slight difference in the swapping criteria. In this case the rx and ry values in the if expression switch places. The swapping mechanic is illustrate in Figure 4.1.

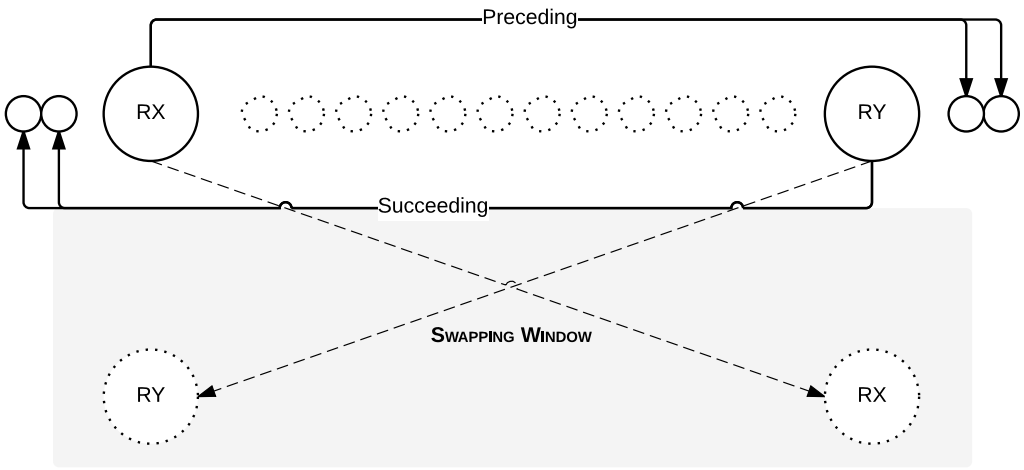

Figure 4.1: How the algorithm re-order rules

Once the algorithm has found a valid swapping window and thus knows that rx and ry can be swapped without breaking the policy integrity, it will do a simple comparison of the rule's matching probability in order to decide whether they should swap or not. However, if the algorithm finds that they should be swapped, then the algorithm won't properly swap them yet, instead the algorithm will find a delta value, Δ_{new}.

The value is created using the matching probability and position number of the rules being compared against and simply gives the estimated average matching time before and after swapping rx and ry, and cn be said to represents the swapping rank of ry. The higher the swapping rank, the more optimal the swap is considered. Consequently, the algorithm will then perform a test to check whether this Δ_{new} value is grater than the current Δ_{max} value. If it is then Δ_{max} is set to this rules Δ_{new} value and then this rule is now the optimal swapping option.

When the inner loop has finished its traversal, a check is performed in order to find if rx should be swapped with a rule or not. If it should be swapped, then the rule with highest delta value, δ_{max}, is chosen as the optimal rule to swap with. Finally, the outer loop will complete the iteration and move on to the next rule at which point the process above is repeated for that rule.

In essence, what this heuristic algorithm tries to achieve, is to get as many rules with a high matching probability as close to the top of the firewall as possible.

4.2.2 The implementation

The actual implementation of the algorithm requires two files to be read by the program which is written in Python. The first file contains the dependency relationships and the initial packet matching probabilities of each rule in the firewall. The second file is an IPtables script containing the actual firewall rules.

The algorithm assume that the IPtables file is consistent, this means that there are no shadowing anomalies in the policy and that the policy in general is a valid configuration without any breaks in the integrity. Examples of both files are given in Listings 4.4 and Listings 4.5.

```
1 name,probability,precedence_relationships
2 A,0.125,B,C
3 B,0.125,D
4 C,0.125,
5 D,0.125,
6 E,0.125,G,H
7 F,0.125,G
8 G,0.125,
9 H,0.125,
```

Listing 4.4: "A small dependency file example"

```
1 iptables -A FORWARD -i eth0 -p udp -s 190.0.0.0/8 --dport 90 -m
      state --state NEW -j ACCEPT -m comment --comment "A"
2 iptables -A FORWARD -i eth0 -p udp -s 190.1.1.0/24 --dport 90:94 -
      m state --state NEW -j DROP -m comment --comment "B"
3 iptables -A FORWARD -i eth0 -p udp -s 190.1.2.0/24 -m state --
      state NEW -j DROP -m comment --comment "C"
4 iptables -A FORWARD -i eth0 -p udp -s 190.1.1.2 --dport 99 -m
      state --state NEW -j ACCEPT -m comment --comment "D"
```

Listing 4.5: "A small sample of an IPtables firewall script"

Each rule in the firewall is stored as an object containing the necessary information for the rule ordering algorithm, Listings 4.6 shows the rule class. Initially, the program reads the dependency file and stores the relevant information from the file into objects. The position attribute in the object is set based on the order in which each rule is read by the program.

Furthermore, each rule object is then stored in a dictionary data structure using the the unique name attribute as the key. Because a dictionary in python is essentially a hash-map, this ensures fast access time to the rules object by simply using the unique name to look up a rule.

```
class Rule:
   def __init__(self, name, rule, prob, pos, preceding, succeeding):
      self.name = name
      self.rule = rule
      self.prob = prob
      self.pos = pos
      self.preceding = preceding
      self.succeeding = succeeding
```

Listing 4.6: "The Rule Class"

However, not all information is attainable from the dependency file, more specifically, the `rule` and `succeeding` attributes can not be taken from the dependency file.

The `rule` attribute contains the actual IPtables command that enables the rule. The attribute is set when the firewall script is read, and in order to match the correct IPtables rule with the existing rules object, the program contains a function which is able to use a regular expression in order to obtain the unique identifier from the comment part of each rule. The function in question is given in Listings 4.7.

```
def get_id(rule):
   pattern = "--comment \"([\w+])\""
   regexp_obj = re.search(pattern, rule)
   if regexp_obj:
      return regexp_obj.group(1)
```

Listing 4.7: "A function that gets the unique identifier of a rule"

The `succeeding` and `preceding` attributes are the same as defined above, namely, lists containing rules that must appear before this rule and rules that must appear after this rule respectively. While the `preceding` attribute was set during the initial reading of the dependency file, the `succeeding` attribute is set at a later point. The rule ordering program contains a function which is able to set each rule's `succeeding` list, the function is given in Listings 4.8.

```
def set_succeeding(rules):
   for rule in rules:
      rule_name = rules[rule].name
      if rules[rule].preceding:
         for r in rules[rule].preceding:
            rules[r].succeeding.append(rule_name)
```

Listing 4.8: "A function that sets each rules succeeding list"

The function will take a rule, r_i and check it against every other rules preceding list, if a match is found in one rules preceding list, then that rule is put in the r_is succeeding list.

Programatically, the rule ordering algorithm pseudocode, given in Algorithm 4.1, has been translated into Listings 4.9.

```
1  def swapping(rules):
2     for rx in rules:
3        delta_max = 0
4        delta_max_rule = None
5        for ry in rules:
6           delta_new = 0
7           if rx != ry:
8              if rules[rx].pos < rules[ry].pos:
9                 rx_preceding_min_pos = find_min(rules, rx,
                      rules[rx].preceding)
10                ry_succeeding_max_pos = find_max(rules, ry,
                      rules[ry].succeeding)
11                if rules[rx].pos > ry_succeeding_max_pos and rules[ry].pos
                      < rx_preceding_min_pos:
12                   if rules[rx].prob < rules[ry].prob:
13                      delta_new = (rules[ry].prob - rules[rx].prob) *
                            (rules[ry].pos - rules[rx].pos)
14                      if delta_max < delta_new:
15                         delta_max = delta_new
16                         delta_max_rule = ry
17             else:
18                ry_preceding_min_pos = find_min(rules, ry,
                      rules[ry].preceding)
19                rx_succeeding_max_pos = find_max(rules, rx,
                      rules[rx].succeeding)
20                if rules[ry].pos > rx_succeeding_max_pos and rules[rx].pos
                      < ry_preceding_min_pos:
21                   if rules[ry].prob < rules[rx].prob:
22                      delta_new = (rules[rx].prob - rules[ry].prob) *
                            (rules[rx].pos - rules[ry].pos)
23                      if delta_max < delta_new:
24                         delta_max = delta_new
25                         delta_max_rule = ry
26       if delta_max > 0:
27          swap(rules, rx, delta_max_rule)
28    return delta_max
```

Listing 4.9: "The rule re-ordering algorithm implementation"

The rule re-ordering function, called swapping here, takes as input, a dictionary of rule objects. The function has two main loops and for every pass of the outer loop, the inner loop traverses the entire rules dictionary. The reason for this is that the algorithm will compare a rule, *rx* from the outer loop with every rule, *ry*, from the inner loop.

Furthermore for each iteration of the outer loop a `delta_max` variable and a `delta_max_rule` variable are initialised to *Zero* and *None* respectively. And for every iteration of the inner loop, a variable, `delta_new` is initialised to *Zero*.

The purpose of these variables are to calculate each rules swapping rank and to find the rule with the highest swapping rank. That rule is then the most optimal rule to perform a swap with. `delta_max_rule` is simply a variable that stores the current optimal swapping rule corresponding to the current delta_max value.

Once the above mentioned variables are initialised, the function will try to find the swapping window by using the functions `find_min` and `find_max` and storing their return values in variables. The function `find_min` will try to find the rule with the lowest position number and

thus higher up in the firewall from the list of rules given as input. It will then return the position number. If the input list is empty, then the function will return the amount of rules in the rules list incremented by 1, as the position number.

The function `find_max` will try to find the rule with the highest position number and thus lower down in the firewall from the list of rules given as input. It will return the position number of the rule it found; if the input list is empty, then the function will return -1. The code for both functions is given in Listings 4.10.

```
1  def find_max(rules, rule, lst):
2     if lst == []:
3        return -1
4     max_pos = 0
5     for r in lst:
6        r_pos = rules[r].pos
7        if max_pos < r_pos:
8           max_pos = r_pos
9     return max_pos
10
11 def find_min(rules, rule, lst):
12    if lst == []:
13       return len(rules) + 1
14    min_pos = len(rules) + 1
15    for r in lst:
16       r_pos = rules[r].pos
17       if min_pos > r_pos:
18          min_pos = r_pos
19    return min_pos
```

Listing 4.10: "The find_min and find_max functions"

When the swapping window is found, a check to find the validity of the window is performed. If the check evaluates to *True*, then a simple comparison of the rules probability is performed in order to decide whether to swap the rules or not. If the function decides that the rules can be swapped, then the rules `delta_new` is calculated based on each rules probability and current position. And if the delta value is greater than the current `delta_max` then the `delta_max` value is updated along with the `delta_max_rule`.

Finally, once the inner loop is finished, a check is performed in order to see if the `delta_max` value is above *zero*, if it is then that means that there is a swappable rule. If the check is evaluated to *True*, then a swap is performed using the rule with the highest delta value. When the outer loop has finished, the function will return the value of the `delta_max` variable.

4.3 Traffic Aware Algorithm

The approach section called for a traffic aware algorithm which is able to dynamically update a rules matching probability as the state of the network

traffic changes in the dynamic network. This section contains the results that satisfies the criteria.

4.3.1 The Algorithm Design

The algorithm is a modified *weak estimator* algorithm [21]. It is modified such that it is able to use a batch of packet matches in order to calculate the packet matching probabilities for a given rule. This ensures that the algorithm does not have to constantly update for each incoming packet. The pseudocode is given in Algorithm 4.13

Algorithm 4.2: The Weak Estimator algorithm

Data: A list of firewall rules, and a lambda value
Result: Updated probabilities for each rule in the list of rules

```
1 for rule in rules do
2     rule.prob = P̂ + λ(M_i/M − P̂)
```

The algorithm takes as input a list of rules and a λ value. It will then iterate through the list of rules and update each rules probability by using the modified weak estimator algorithm. In order to update each rules probability, the algorithm calculates it using the previous probability of the given rule, $\hat{P}$, the total amount of packet matches, M and the amount of packets matches for one rule, M_i.

4.3.2 The implementation

The actual implementation of the weak estimator algorithm requires a file containing a unique identifier and initial probability for a rule as input, this has been solved by utilising the same dependency file given in Listing 4.4, that the rule ordering algorithm uses. The difference in the file reading function is simply that the weak estimator file reader only returns a dictionary with the unique identifier as key and the initial probability as value.

In order for the weak estimator function to get up to date packet matching values, a function which is able to read the iptables statistics was needed. The function is given in Listing 4.11.

```
def read_iptables_stats():
   process = subprocess.Popen(['iptables', '-xnvL'],
       stdout=subprocess.PIPE)
   out, err = process.communicate()
   out = out.splitlines()
   pkts_per_rule = {}
   for line in out:
      pkts, name = get_pkts(line)
      if pkts and name:
         pkts_per_rule[name] = int(pkts)
   return pkts_per_rule
```

Listing 4.11: "A function that reads the IPtables in order to get the amount of packet matches for a rule"

The function uses the subprocess module in Python in order to run an IPtables command that produces statistics about the current firewall. The output of the command is stored in a list, `out`. For every iteration of the list `out`, each line (containing statistics about one rule) is given to the function `get_pkts`. The `get_pkts` function is given in Listing 4.12 and uses a regular expression to get the packet count and unique identifier from the input rule.

```
def get_pkts(rule):
   pattern = "^\s*([\d]+).* /\* ([\w]+) \*/"
   regexp_obj = re.search(pattern, rule)
   if regexp_obj:
      return regexp_obj.group(1), regexp_obj.group(2)
   else:
      return None, None
```

Listing 4.12: "A function that uses a regular expression to get the number of packet matches and the unique identifier for a rule"

The return value from `get_pkts` is stored in a dictionary with the identifier as a key and packet count as value. Once the iteration over `out` is completed, the resulting dictionary is returned.

Programatically, the weak estimator algorithm pseudocode, given in Algorithm 4.13, has been translated into Listing 4.13.

```
1  def weak_estimator(rules, lmbda):
2     pkts_per_rule = read_iptables_stats()
3
4     M_old = 0
5     for v in rules.values():
6        M_tmp = v[1]
7        M_old += M_tmp
8     M_new = sum(pkts_per_rule.values())
9     M = M_new - M_old
10
11    if M > 0:
12       for rule in rules:
13          cur_prob = rules[rule][0]
14          cur_pkts = rules[rule][1]
15          upd_pkts = pkts_per_rule[rule]
16          M_i = float(upd_pkts - cur_pkts)
17          upd_prob = cur_prob + (lmbda * (M_i/M - cur_prob))
18          rules[rule] = [upd_prob, upd_pkts]
```

Listing 4.13: "The weak estimator algorithm implementation"

The functions starts by getting updated packet matching values for each rule by calling the `read_iptables_stats` function. Using the updated values, a variable, M_{new}, is created and contains the total sum of the number of all the updated packet matches for each rule. The function will then calculated all the total number of packet matches up till this point and store the value in the variable, M_{old}.

Moving on, the function will subtract M_{old} from M_{new}, this will give the total amount of packet matches since the last time this function was called, the new value is stored in the variable `M`.

Finally, a check will be performed, testing whether or not there has been any new packet matches since the last time this function was called, thus if `M` is *zero* or less, the function will simply finish. If `M` is grater than *zero* then the function will start to iterate over the list of rules and for each rule, calculate and update it with its new probability.

4.4 Firewall Optimiser

The approach section called some functionality in order to combine the rule re-ordering algorithm and the traffic aware algorithm such that they are able to communicate between them selves. The traffic aware algorithm must also be able to update the probability of a rule and have the same effect reflected in the rules used by the rule re-ordering algorithm. This section contains the results that satisfies the criteria.

4.4.1 The Implementation

In order for the algorithms to communicate between each other, each algorithms necessary data structures must be initialised, the init function in

the rule ordering program was created with this in mind, the code is given in Listings 4.14.

```
1
2  def init(filename_dep, filename_fw):
3     filename = filename_dep
4     iptables_filename = filename_fw
5
6     rules = read_DAG(filename)
7     iptables_preamble, iptables_rules = read_iptables(iptables_filename)
8
9     for rule in iptables_rules:
10       rid = get_id(rule)
11       rules[rid].rule = rule # set the rules field in the relevant object
12
13    set_succeeding(rules)
14
15    return rules, iptables_preamble
```

Listing 4.14: "Initialising the rule ordering structures"

The function takes as input two filenames, the first is for a dependency file containing the dependency relationships and initial packet matching probabilities. The second file is an IPtables script containing the firewall policy. Examples of both files are given in Listings 4.4 and Listings 4.5. The function will then call functions for reading the two files, the first read function will create objects out of the information in the dependency file and store them in a dictionary called `rules`.

The second read function will create two lists called `iptables_preamble` and `iptables_rules`, the former contains the preamble of the IPtables script such as *ssh* rules that must always be there in order to ensure connectivity to the machine. The latter contains the IPtables commands for applying the actual firewall. Once the data structures are initialise, the init function will set the `rule` attribute in the rule object and populate each rule's succeeding list.

In order to update a rule object based on updated probability values from the traffic aware algorithm, the function in Listing 4.15 was created.

```
1  def rule_update(preamble, rules_obj, rules):
2     body = []
3     # simply overwrite the value list with an updated value list
4     for r in rules:
5        rules_obj[r].prob = rules[r][0]
6        tmp_lst = rules[r]
7        tmp_lst.pop()
8        tmp_lst.append(0)
9        rules[r] = tmp_lst
10
11    for i in range(swp_num):
12       delta_max = rule_ordering.swapping(rules_obj)
13       if delta_max <= 0:
14          break
15
16    for r in (sorted(rules_obj.values(), key=operator.attrgetter('pos'))):
17       body.append(r.rule)
18    filename = rule_ordering.write_firewall(preamble, body)
19
20    process = subprocess.Popen(['bash', filename], stdout=subprocess.PIPE)
21    out, err = process.communicate()
```

Listing 4.15: "The rule update function"

The function takes as input the IPtables script preamble, a dictionary of rules objects, and a dictionary of rule identifiers and probabilities. Because IPtables will reset the packet count of all rules once a firewall is re-applied, this must also be reflected in the code. Lines 4 to 9 in Listing 4.15 reflects this.

Once the rules dictionary is updated to reflect the changes made by IPtables, the function will execute the rule ordering algorithm by calling the swapping function. The `swp_num` variable represents the amount of times the program will call the the rule ordering swapping function on a ruleset. Essentially, the higher the number the more optimal the rule re-ordering will be, and for a policy with a large amount of rules, a higher `swp_num` should produce the best results.

Moving on, for each call to the rule ordering swapping function, a check is performed on the `delta_max` return value. If the value is *zero* or less, it means that there was no rule swapping in the last call to the swapping function, which means that there is no reason to continue calling the swapping function and one can instead exit the loop earlier, slightly decreasing the time taken to re-order the policy.

When the policy has been re-ordered, the function will iterate over the `rules_obj` dictionery, sorted by each rule objects position number and append each rule objects actual rule into the `body` list. The list will then contain a valid ordering of firewall rules. The function will then call a new function for writing the firewall into a file. The function takes the IPtables preamble and the body list as input.

Once the new firewall configuration has been written to a file, the rule update function will use the subprocess module to apply the newly created firewall script.

Finally, the main function of the optimiser is given in Listing 4.16.

```
def main():
   filename = args.filename
   filename_fw = args.firewall

   rules_obj, iptables_preamble = rule_ordering.init(filename,
       filename_fw)
   rules = read_DAG(filename)
   initial_pkts_per_rule = read_iptables_stats()
   lmbda = 0.2
   for i in initial_pkts_per_rule:
      cur_pkts = initial_pkts_per_rule[i]
      tmp_lst = rules[i]
      tmp_lst.pop()
      tmp_lst.append(0)
      rules[i] = tmp_lst

   try:
      while True:
         weak_estimator(rules, lmbda)
         rule_update(iptables_preamble, rules_obj, rules)
         time.sleep(wait_time)
   except KeyboardInterrupt:
      print "exiting the program"
      exit(0)
```

Listing 4.16: "The main function"

The main function will initially, simply initiate the necessary data structures for both the rule ordering algorithm and the traffic aware algorithm. When the the initialisation is completed, the function will enter the main loop.

As can be seen in Listing 4.16, the update function is called immediately after the weak estimator function has updated its values. However, as can be seen in the dynamic experiments described in the approach section, the conditions for calling the rule update function can be changed to suite a users needs.

4.5 Proofs

In this section, we present some theoretical results related to our work. The first result concerned the optimality of the devised Generalised Weak Estimator that we propose in this paper. The algorithm is a generalisation of the Stochastic Learning Weak Estimator proposed by Oommen and Rueda [21]. The main difference is that the Stochastic Learning Weak Estimator operates in an incremental manner, i.e, updates the probability estimates upon receiving an "individual observation". The Generalised Stochastic Learning Weak Estimator that we propose in this thesis is able to handle a batch of M observations.

The second result that we prove is related to the condition that we use for swapping two rules, namely Δ_{new}. We will show that Δ_{new} is simply the difference of the average matching time before and after swapping.

4.5.1 The Generalised Weak Estimator

Specifically, let X be a multinomially distributed random variable, which takes on the values from the set $\{'1', \ldots, 'r'\}$. We assume that X is governed by the distribution $S = [s_1, \ldots, s_r]^T$ as follows:

$X =$ 'i' with the probability s_i, where $\sum_{i=1}^{r} s_i = 1$.

We assume that between each discrete time instants n and $n+1$, we obtain a batch of M concrete realisations of X.

Let $\{x(n,1), x(n,2), x(n,3), ..., x(n,M)\}$ denote the batch of M observations obtained between the time instants n and $n+1$.

The intention of the exercise is to estimate S, i.e., s_i for $i = 1, \ldots, r$ based on the batch of observations. We achieve this by maintaining a running estimate $P(n) = [p_1(n), \ldots, p_r(n)]^T$ of S, where $p_i(n)$ is the estimate of s_i at time 'n', for $i = 1, \ldots, r$. We omit the reference to time 'n' in $P(n)$ whenever there is no confusion.

Let $M_i(n)$ be the number of elements in the batch $\{x(n,1), x(n,2), x(n,3), ..., x(n,M)\}$ for which $X =$ 'i'. In more formal terms $m_i(n) = \sum_{k=1}^{M} I(x(n,k) = 1)$ where $I(.)$ is the indicator function. Then, the value of $p_i(n), 1 \leq i \leq r$, are update in the following way:

$$p_i(n+1) \quad \leftarrow \quad p_i(n) + \lambda(\frac{m_i(n)}{M} - p_i(n)) \tag{4.1}$$

The informed reader should note that the above algorithm is just a generalization of Oommen's original weak estimator algorithm [21]. In fact, it is easy to note that whenever $M = 1$, the above updated equation coincides with the original algorithm devised by Oommen and Rueda [21].

The properties of the estimator are catalogued below.

Theorem 1 *Let the parameter S of the multinomial distribution be estimated by $P(n)$ at time 'n' as per equation (4.1). Then, $E[P(\infty)] = S$.*

Proof The expected value of $p_i(n+1)$ given the estimated probabilities at time 'n', P, is:

$$
\begin{aligned}
\mathrm{E}\left[p_i(n+1)|P\right] &= [p_i(n) + \lambda(\frac{k}{M} - p_i(n))] \sum_{k=0}^{M} Prob(m_i(n) = k) && (4.2)\\
&= (1-\lambda)p_i + \lambda \sum_{k=0}^{M} \frac{k}{M} \binom{M}{k} s_i^k (1-s_i)^{M-k} && (4.3)\\
&= (1-\lambda)p_i + \lambda \sum_{k=0}^{M} \frac{k}{M} \frac{M!}{k!(M-k)!} s_i^k (1-s_i)^{M-k} && (4.4)\\
&= (1-\lambda)p_i + \lambda \sum_{k=1}^{M} \frac{(M-1)!}{(k-1)!(M-k)!} s_i^k (1-s_i)^{M-k} && (4.5)\\
&= (1-\lambda)p_i + \lambda s_i \sum_{k=1}^{M} \binom{M-1}{k-1} s_i^{k-1} (1-s_i)^{M-k} && (4.6)\\
&= (1-\lambda)p_i + \lambda s_i \sum_{l=0}^{M} \binom{M}{l} s_i^{l} (1-s_i)^{M-l} && (4.7)\\
&= (1-\lambda)p_i + \lambda s_i && (4.8)
\end{aligned}
$$

In Equation (4.3) we apply the mulinomial distribution theorem in order to obtain $Prob(m_i(n) = k)$. In Equation(4.7), we apply a change of the variable k, where $k - 1 = l$. While in Equation(4.8), we apply the binomial theorem.

Taking expectations a second time, we have

$$
\mathrm{E}[p_i(n+1)] = \lambda s_i + (1-\lambda)\mathrm{E}[p_i(n)]\,. \tag{4.9}
$$

As $n \to \infty$, both equations $\mathrm{E}\left[p_i(n+1)\right]$ and $\mathrm{E}\left[p_i(n)\right]$ converge to $\mathrm{E}\left[p_i(\infty)\right]$, and can be written:

$$
\begin{aligned}
\mathrm{E}[p_i(\infty)]\lambda &= \lambda s_i && (4.10)\\
\Rightarrow \mathrm{E}[p_i(\infty)] &= s_i\,. && (4.11)
\end{aligned}
$$

The result follows because (4.11) is valid for every component p_i of P. The following result follows directly from Oommen and Rueda [21].

Theorem 2 *Thus the rate of convergence of P is* fully *determined by λ.*

4.5.2 The Swapping Condition

Theorem 3 *The difference of the average matching time before swapping and after swapping two rules rx and ry is given by: $\Delta_{new} = (ry.prob - rx.prob).(ry.pos - rx.pos)$*

Let $rk.pos$ be the position of rule k before swapping rx and ry, and let $rk.pos'$ be the position of rule k after swapping rx and ry.

It is easy to note that:

- $rk.pos = rk.pos'$ if $k \neq x$ and $k \neq y$, and that
- $rx.pos' = ry.pos$
- $ry.pos' = rx.pos$

$$
\begin{aligned}
\Delta_{new} &= \text{The Average Before Swapping} - \text{The Average After Swapping} \\
&= \sum_{k=1}^{N} rk.prob \cdot rk.pos - \sum_{k=1}^{N} rk.prob' \cdot rk.pos \\
&= (rx.pos\ rx.prob + ry.pos \cdot ry.prob) - (rx.pos' \cdot rx.prob + ry.pos' \cdot ry.prob) \\
&= (rx.pos\ rx.prob + ry.pos \cdot ry.prob) - (ry.pos \cdot rx.prob + rx.pos \cdot ry.prob) \\
&= rx.pos(rx.prob - ry.prob) + ry.pos(ry.prob - rx.prob) \\
&= (ry.prob - rx.prob) \cdot (ry.pos - rx.pos)
\end{aligned}
$$

Note that $\Delta_{new} = (ry.prob - rx.prob) \cdot (ry.pos - rx.pos) = (rx.prob - ry.prob) \cdot (rx.pos - ry.pos)$

Chapter 5

Result and Analysis 2: Experiments

This chapter contains the results of the experiments as described in the approach section.

5.1 Static Experiment 1: Intra-rule re-ordering

As described in the approach section, this experiments goal was to show that the rule ordering algorithm is able to re-order rules while maintaining the integrity of the firewall policy. Figure 5.1 shows the initial conditions as outlined in the approach.

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
      18      504 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
      19      532 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
      87     2436 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
      26      728 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
       6      240 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
      10      400 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
      10      400 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
       5      200 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
```

Figure 5.1: The FORWARD chain of iptables containing the firewall rules for experiment 1

Observing figure 5.2, it is apparent that the C rule has been moved above rule B but below rule A. This is expected as there is no dependency relationship between rule B and C, but there is one between rules A and C which is why rule C must be placed underneath it for the policy integrity

to be maintained. The average matching time calculates to 2.6535, which is the same as the expected value.

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
       0        0 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
       0        0 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
       0        0 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
       0        0 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
       0        0 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
       0        0 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
```

Figure 5.2: The FORWARD chain after rule re-ordering

5.2 Static Experiment 2: Inter-rule re-ordering

This experiments intent was to prove that the rule ordering algorithm is able to re-order non-dependent rules while maintaining the policy integrity of the firewall. Figure 5.1 shows the initial conditions as outlined in the approach.

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
      11      308 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
       9      252 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
      11      308 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
       9      252 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
      77     3080 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
      38     1520 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
      16      640 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
      12      480 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
```

Figure 5.3: The FORWARD chain of iptables containing the firewall rules for experiment 2

The results shown in figure 5.4 were largely the same as the expected results, the rules E - H are at the top as expected while the rules A - D are at the bottom. The one difference from the expected results is that rule C is above rule B rather than the expected order of A, B, C and D.

However this is still a valid result as the intent was to observe the re-ordering of non-dependent rules, thus, the intra-rule re-ordering has no bearing on the outcome of the experiment. The average matching time

calculates to 2.6619, which is slightly worse than the expected value of 2.6535. The reason for this is that there were more packet matches for rule C than there were for rule B, despite rule B having the superior probability.

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               destination
       0        0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0           state RELATED,ESTABLISHED
       0        0 ACCEPT     tcp  --  *      *       0.0.0.0/0            192.168.128.206     tcp dpt:22 state NEW
       0        0 ACCEPT     tcp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           tcp dpt:90 state NEW /* E */
       0        0 DROP       tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpt:88 state NEW /* F */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           tcp dpts:88:94 state NEW /* G */
       0        0 ACCEPT     tcp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* H */
       0        0 ACCEPT     udp  --  eth0   *       190.0.0.0/8          0.0.0.0/0           udp dpt:90 state NEW /* A */
       0        0 DROP       udp  --  eth0   *       190.1.2.0/24         0.0.0.0/0           state NEW /* C */
       0        0 DROP       udp  --  eth0   *       190.1.1.0/24         0.0.0.0/0           udp dpts:90:94 state NEW /* B */
       0        0 ACCEPT     udp  --  eth0   *       190.1.1.2            0.0.0.0/0           udp dpt:99 state NEW /* D */
```

Figure 5.4: The FORWARD chain after rule re-ordering

5.3 Static Experiment 3: Comparing against Fulps simple algorithm for rule re-ordering

As described in the approach section, this experiments goal was to compare our rule ordering algorithm with that of Fulps rule ordering algorithm and observe the difference in average matching time.

In the the program that created the Direct Acyclic Graphs there is a variable that decides the percentage chance that a pair of rules will have a dependency relationship between them. This experiment set the value to 1% and 5%. The results of the experiments are shown in Table 5.1 and Table 5.2 for 1% and 5% respecitvely.

Percentage	Initial	Our Algorithm	Fulp	Number of Rules
1 %	26,42	12,42	24,157	100
1 %	41,18	12,16	38,81	100
1 %	31,657	12,12	24,45	100
1 %	37,31	10,87	36,2	100
1 %	37,56	11,37	35,011	100

Table 5.1: The result from the comparison with the DAG having a 1% chance of an edge between two nodes

Table 5.1 shows the result of five tests done using a 1% chance of edges on a graph with a 100 nodes (or rules). Initially, we notice that Fulps algorithm is not able to improve the average matching time by a lot. Compared to

our algorithm which is able to significantly improve the average matching time.

Comparing the averages of the values, we find that Fulps algorithm, with an average of 31.7256, is able to improve the initial average matching time, with an average of 34.8254, by 8.901%. While our algorithm, with an average of 11.788, improved it by 66.1511% which is 62.8439% better than Fulps algorithm.

Percentage	Initial	Our Algorithm	Fulp	Number Rules
5 %	41,98	22,87	41,33	100
5 %	30,28	21,321	29,06	100
5 %	46,371	27,701	45,52	100
5 %	33,7	21,71	33,115	100
5 %	59,12	28,06	58,36	100

Table 5.2: The result from the comparison with the DAG having a 5% chance of an edge between two nodes

Regarding able 5.2, the results state largely the same and comparing the averages of the values, we find that Fulps algorithm, with an average of 41.477, is able to improve the initial average matching time, with an average of 42.2902, by 1.9229%. Ours, with an average of 25.5324, on the other hand is able to improve it by 39.6257%, which is an 38.442% improvement over Fulps algorithm.

However, overall, while we found our algorithm to be significantly better than Fulps algorithm. We also noticed that the denser the firewall policy (DAG) is, the harder it is to find an optimal ordering. This is evident by observing that the average matching time increased significantly when increasing the chance of an edge between to nodes, and thus making the graph denser.

5.4 Dynamic Experiment 1: Schedule based rule re-ordering with dynamic traffic

The intention of this experiment was to test both the rule order and traffic aware algorithms using a schedule based re-ordering policy in a dynamic network. In order to do so we had to change the main loop of the firewall performance optimiser to be able to use a schedule based policy. The code for this is given in Listing 5.1.

```
1  glob_pkt_cnt_thrsh = 100
2
3  try:
4        while True:
5           weak_estimator(rules, lmbda)
6           if glob_pkt_cnt_cur >= glob_pkt_cnt_thrsh:
7              rule_update(iptables_preamble, rules_obj, rules)
8              glob_pkt_cnt_cur = 0
9
10          if glob_pkt_cnt_tot >= 1000:
11             calc_average_match_time(rules, rules_obj, rules_zipf_Y,
                   static_fw, log_filename)
12          else:
13             calc_average_match_time(rules, rules_obj, rules_zipf_X,
                   static_fw, log_filename)
14
15          time.sleep(2)
16    except KeyboardInterrupt:
17       print "exiting the program"
18       exit(0)
```

Listing 5.1: "Main loop for the schedule based policy"

Moving on, the resulting graphs for this experiment are given in Figure 5.5 and Figure 5.6.

The first experiment results appear to be behaving as expected. We observe that both the **True** and **Estimated** average matching times start with high values, indicative of a poorly optimised rule ordering, before gradually improving their times. However, there are some fluctuates in the results causing spikes.

These might be because of the nature of the traffic generator as it wont guarantee that packets with high probabilities are always chosen, as the generator uses a roulette wheel function in order to decide which rule to test, we might end up with low probability rules being chosen at random and thus being sent to the firewall producing the fluctuations we observer.

When comparing the **True** and **Estimated** average matching times, we observe that they match relatively closely, with the **Estimated** values being consistently slightly below the true average matching time.

Moreover, the base line time behaves as expected, it starts with a high average packet matching time until the switch, at which point it sharply decreases and has a relatively optimal matching time.

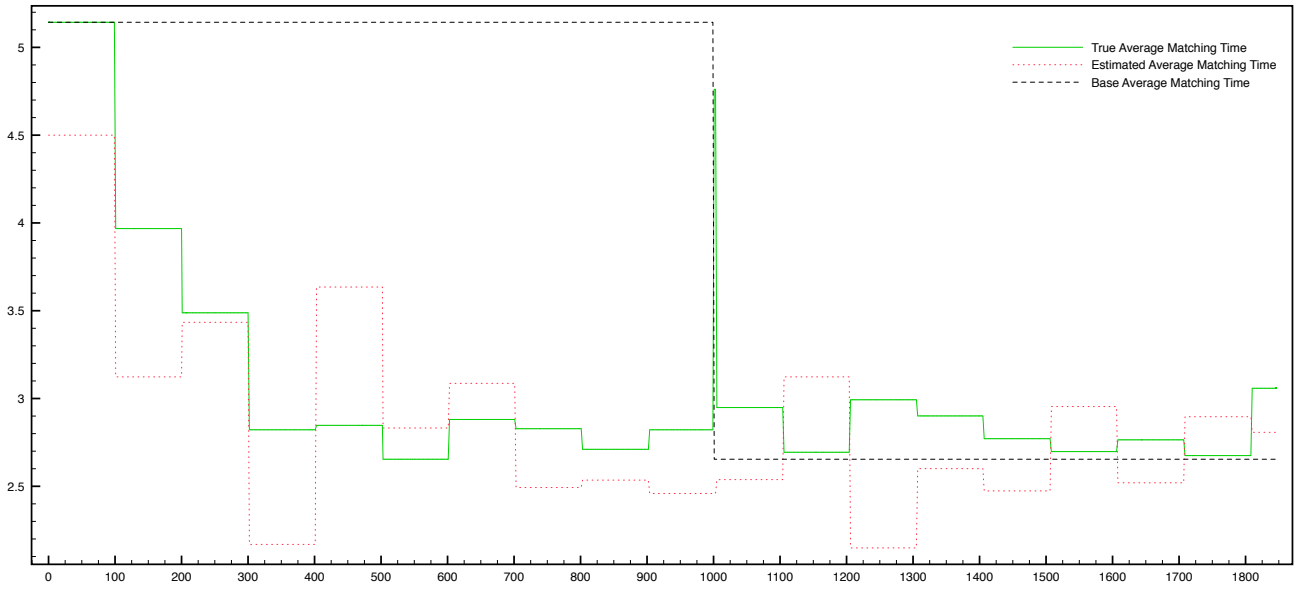

Figure 5.5: The first experiment results

The second experiment results are given in Figure 5.6.

These results produces more unexpected results as there seems to have been more fluctuations or lack thereof. The base line time is working as expected, however, the **True** and **Estimated** times seem to be too flat. Again this could simply be due to bad luck with the traffic generator.

We also observe that the **True** and **Estimated** times don't match relatively closely anymore, the reason for this might be because of the generally low updates to each rule as it matches a packet given by the weak estimator function.

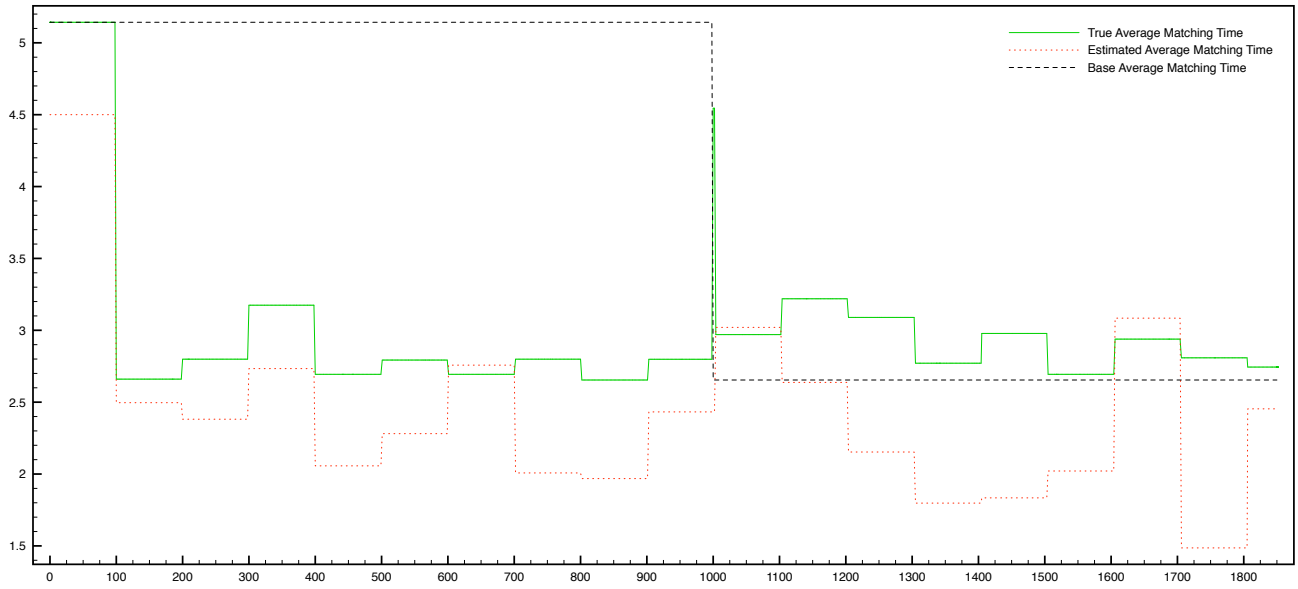

Figure 5.6: The second experiment results

5.5 Dynamic Experiment 2: Extended Schedule based rule re-ordering with dynamic traffic

This experiment is very similar to the previous experiment and the intention was to observe the long term effects of the algorithms. Thus, we must update the following variables in the Listing 5.1 to the one below, for the code to be valid for this experiments:

1. `glob_pkt_cnt_thrsh` must be set to 1000
2. `glob_pkt_cnt_tot` must be set to 10000

The results of this experiment are given in the graph in Figure 5.7

As can be seen, the base line is within the expectations. It has a low average matching time in the beginning and once the switch occurs at 10000 packet its average matching time increases sharply and becomes very poor.

Moving on, we observe that the **True** average matching time is behaving unexpectedly as it should see a sharp increase in average matching time once the switch occurs. However, comparing the **True** and **Estimated** times before the distribution switch, we observe that they behave very similarly. The only difference being that the **True** matching time has a much higher average matching time than the **Estimator**. This can be explained by the fact that our version of the weak estimator only increases each rule by a small value

Furthermore, we notice that the **Estimated** time seems to be behaving as expected, with the exception that it has a consistently lower average

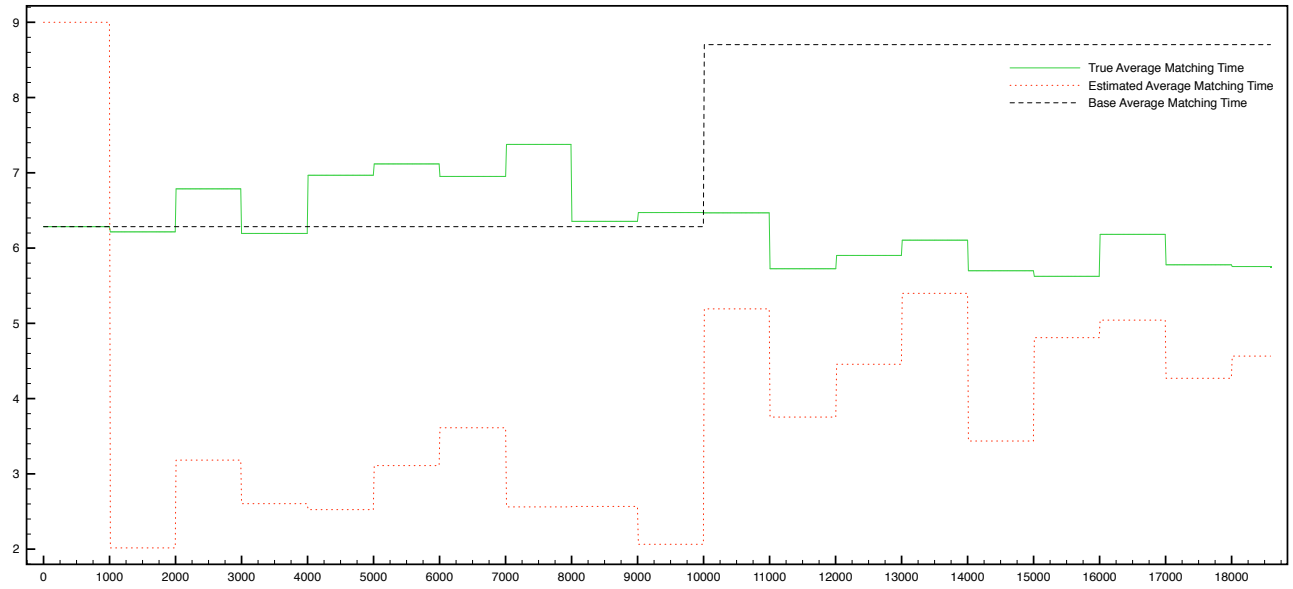

Figure 5.7: The graph shows the result from the extended schedule based experiment

matching time, which can be explained in the same manner as above, that our version of the weak estimator algorithm increases and decreases the rule matching probabilities by a small amount each time.

5.6 Dynamic Experiment 3: Performance triggered rule re-ordering using a sliding window

The intention of this experiment was to observe the performance of the algorithms when using a performance triggered condition. In order to test this, the code had to be adapted so that it could use such a condition. The code is given in Listing 5.2.

```
try:
      while True:
         weak_estimator(rules, lmbda)
         tmp_avg_mtch_tme = calc_average_match_time(rules, rules_obj,
             log_filename)
         length = len(avg_mtch_tme_wndw)
         if length < max_wndw_size:
            avg_mtch_tme_wndw.append(tmp_avg_mtch_tme)
         else:
            # fifo list, first in first out, we only want the lates values
            avg_mtch_tme_wndw.pop(0)
            avg_mtch_tme_wndw.append(tmp_avg_mtch_tme)

      # return true if trend is increasing, else false
         if length == max_wndw_size and find_trend(avg_mtch_tme_wndw):
            rule_update(iptables_preamble, rules_obj, rules)

         time.sleep(2)
   except KeyboardInterrupt:
      print "exiting the program"
      exit(0)
```

Listing 5.2: "Main loop for the performance triggered policy"

The results of this experiments are given in Figure 5.8 and Figure 5.9.

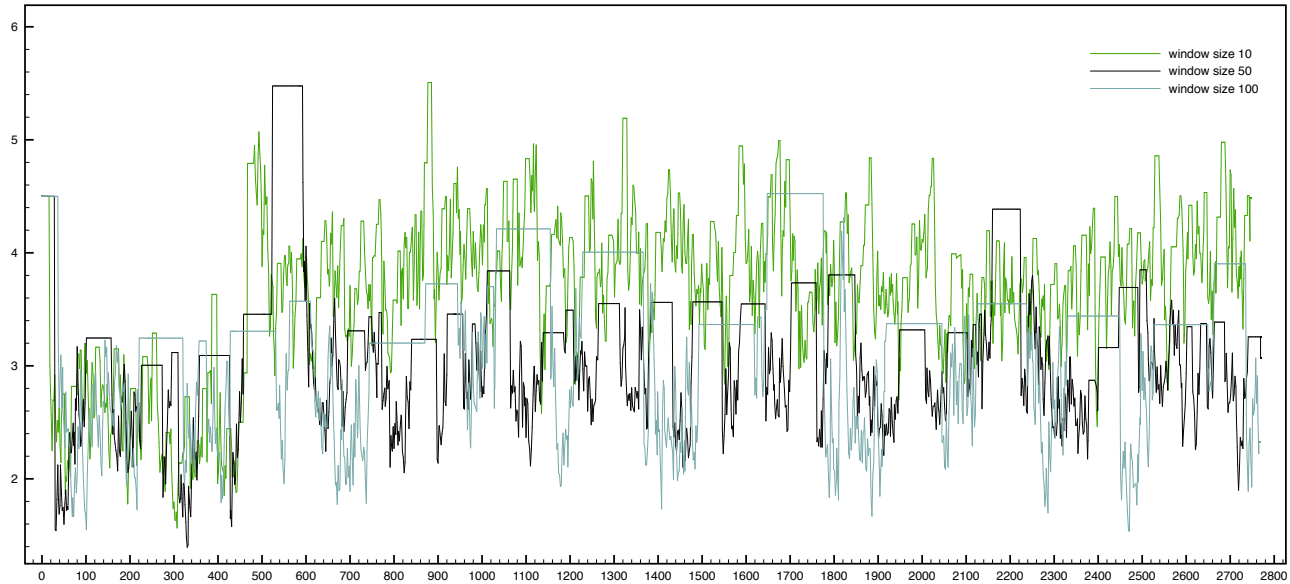

Figure 5.8: The graph for switching distribution every 500 packets sent

From observing the result we see that in general, the bigger the window size, the lower the average matching time is. The reason for this is because we have more information to find if the trend shows an overall increase or decrease in matching time. A small window size will give a lot of false positives resulting in too early rule ordering. We notice that in Figure 5.8, when the window size is 10, the average matching time is consistently higher than for window size equaling 50 and 100.

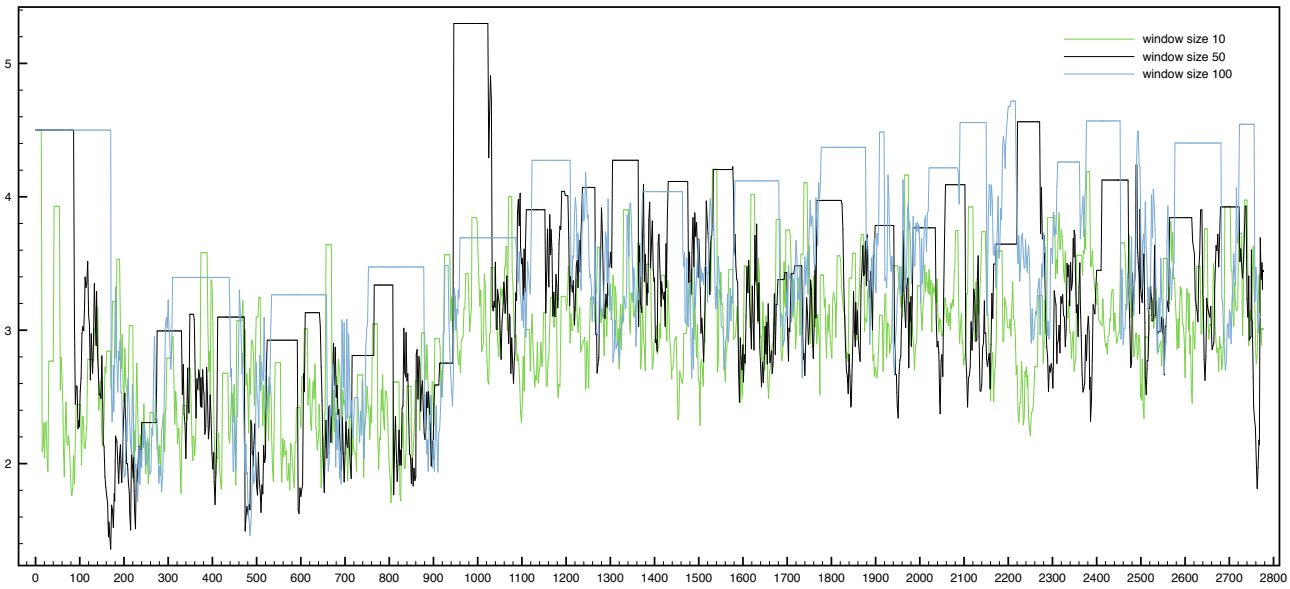

Figure 5.9: The graph for switching distribution every 1000 packets sent

However, in the graph where the distribution was switched after 1000 packets, rather than just 500, we observe that even a window size of 10 is able get comparatively good results relative to a window sizes of 50 and 100. The reason for this is that the network traffic state will stay similar for a longer time until the switch occurs. There are some fluctuations here, but they can be be caused by the random nature of the traffic generator.

Overall, the results match the expected results.

Chapter 6

Discussion

6.1 Project Evaluation and Impact

In this thesis the field of firewall optimisation was evaluated in regards to firewall rule ordering and traffic awareness. Two algorithms were designed, implemented and combined into a proof of concept firewall optimiser. The firewall optimiser was able to re-order rules according to statistics taken from the network about each firewall rule.

The rule ordering algorithm is able to sort a firewall policy based on each rules packet matching probability, dependency relationships and current position in the firewall. This algorithm came about as a combination of the simplicity of Fulp's algorithm and a desire to improve upon its weaknesses. The result was a simple but also a deep algorithm in the sense that it considers many different metrics for rule re-ordering by reducing the average matching time of the firewall.

The traffic aware algorithm is able to update a rules matching probability by reading the IPtables statistics about each rule and then utilise a novel version of the *Weak Estimator* algorithm which is able to use a batch of packet matches to update a rules matching probability as opposed to the original *weak estimator* which requires a *one-by-one* updating policy i.e it must keep track of every packet in the system. The latter is i precisely why the *weak estimator* algorithm using batch came about, due to the fact that we could not efficiently catch a packet one by one.

6.1.1 The experiments

There were a total of six experiments in this thesis, the first two were simple proof of concept experiments with the intention of showing that the rule ordering algorithm was able to re-order rule such that the new rule order was superior to the original and that the policy integrity was kept intact.

The third experiment was comparative a study, where the algorithm by Fulp was compared against our rule ordering algorithm. The intention was to show that our algorithm was able to more efficiently optimise a given firewall policy. The results found that a comparison of our rule ordering algorithm performed significantly better than Fulp's algorithm, by as much as a 68% reduction of the average matching time.

The tree last experiments were more dynamic in nature and were designed and implemented because our rule re-ordering algorithm is slow. The slowness is because it is based on a modified bubble sort algorithm which is an exhaustive search. Thus, in order for it to produce optimal solutions it must be run multiple times and the larger the policy, the more time the algorithm must run. However, this is not feasible, especially in a real life scenario.

In an attempt to reduce the slow and exhaustive nature of the rule ordering algorithm, the aforementioned experiments were created. These experiments contain the proposed solutions to the slowness problem. The first two are schedule based rule re-oredering policy experiments.

A schedule based policy is simply a policy where you specify when an action should be executed, in this case, the action was to call the rule update function. The experiments set the execution policy to be after 100 packets and a 1000 packets respectively.

The resulting graphs show that it could be a viable solution to the slowness problem of the re-ordering algorithm, however due to consistent packet losses, it was hard to interpret the resulting graphs. Section 6.2.1 explains the problem.

The last experiment was a performance triggered rule re-ordering experiment. A performance trigger is similar to schedule based rule re-ordering, however, instead of using simple variables to find when to run the rule update function, it will instead calculate the optimal time to call the update function.

In the experiment, we used a sliding window to contain the latest average matching times of the policy and tried to find the trend in the window. If the trend was an overall increase in matching time then that means that the current rule ordering is not optimal and must be re-ordered.

The results from this experiment reveals that this type of re-ordering works very well. The graphs showed that the average matching time was consistently kept at a low level. However, the problem with performance triggering is that they could potentially become very computationally heavy.

6.1.2 Impact

Primarily, large organisation and institutes will benefit the most from this thesis as they must handle large amounts of data from their workforce. Ensuring that the firewall is optimised is especially important because of the predictions made by the *Internet of Things* [1].

The paper predicts that each person will have 6.58 devices connected to the internet on average, resulting in 50 billion devices being connected to the internet world wide, thus, having an optimised firewall will ensure that the firewall does not become a bottleneck in the current and future high-speed networks.

Furthermore, with so many devices connected to the internet this means that there will invariably be a lot of sensitive data stored on various networks, thus, having an optimised firewall will be able to mitigate network attacks more so than a static and un-optimised firewall.

Finally, the results of this thesis are easy to integrate with existing firewalls such as IPtables and the *Weak Estimator Batch* algorithm that we found should also be usable in every instance that the original is.

6.2 Technical Pitfalls, Limitations and Inaccuracies

During the course of writing this thesis and answering the problem statement various limitations and inaccuracies were discovered regarding the firewall performance optimiser.

6.2.1 An Unstable Environment

During the the experimental phase of this thesis there were several instances were an experiment was dependent on accurate communication between the traffic generator and the firewall optimiser. For instance, during the the dynamic experiments regarding the schedule base rule re-ordering policy, the test was based on calculating the average matching time of the firewall. There were three instances of the average matching time that the experiment wanted to compare. The first was the True average matching time, the second was the estimate average matching time and the last was the base line average matching time.

The latter instance was not the problem as the base line could be calculated without outside interference, however, both the true and estimated average matching time took the current position of a rule into account when calculating the time. At first thought, this should not cause any significant problems as the whole point of the firewall optimiser is to periodically re-order the policy rules, however, the experiment was set up such that after

X amount of packets had been sent from the traffic generator, the generator itself would switch its zipf distribution to another ordering.

The idea was that this would force the optimiser to to do some major rule re-ordering making the resulting graph more interesting. The problem occurred because the true average matching time doesn't use the probability data from the optimiser itself when calculating its average matching time; in order to calculate the true average matching time, the optimiser uses the same distribution as that of the traffic generator, thus, when the generator switches distribution after X amount of packets have been sent, the optimiser must also do the same after a similar amount of packets have been matched. And it is herein the problem lies.

While executing the experiment there was a lot of packet loss on the firewall optimiser side. This packet loss causes the timing of the distribution switch on the firewall optimiser side to be wrong. This in turn caused the true average matching time to be out of sync with the traffic generator and the estimated average matching time.

The reason for this packet loss could be three-fold. It could be a problem with the traffic generator as it is unable to guarantee that all packets sent will reach its destination. The reason for this is that it simply sends SYN packets and doesn't wait for reply. On the other hand the inaccuracies and packet loss might be because of an inherent instability in the cloud environment.

As one of the benefits of a cloud environment is that all the underlying hard ware has been abstracted away in order to simplify the process of creating new instance. However, because of this streamlining one doesn't know everything about the underlying hard ware, this means that one doesn't know whether multiple virtual machine instance are situated on the same hardware or if they are spread across the hardware encompassed by the cloud.

Thus, there might be a lot of hops between the traffic generator and the firewall optimiser, and because the traffic generator does not guarantee the arrival of a packet, the prospect of packet loss due to packet getting lost on the way is something that should be taken into account.

The third possibility for the packet loss might also be that the optimiser setup itself takes too long to re-order rules that by the time the new firewall is applied and the program can continue monitoring the iptables statistics, packets might not have been counted in between. One way to solve this problem would be to make the firewall optimiser system work in parallel, with one thread for rule ordering, one thread for weak estimator updating and a final thread for communication between the algorithms.

Either way more testing is needed in order to find the cause of the packet losses.

6.2.2 Limitations of the Weak Estimator Algorithm

The *Weak Estimator Batch* algorithm created in this thesis is independent of the batch size; recalling that the weak estimator algorithm will use the following formula to calculate an updated probability for a given rule,

$$\hat{p}_i + \lambda(\frac{m_i}{M} - \hat{p}_i)$$

Where M_i is the amount of current packet matches for a given rule and that M is the total amount of current packet matches for the entire firewall policy. It is apparent that for any equal value of M_i and M the algorithm cannot tell the difference between,

$$\frac{m_i = 4}{M = 5} \Leftrightarrow \frac{m_i = 40}{M = 50}$$

It should be able to understand that $\frac{40}{50}$ are more packets than $\frac{4}{5}$ and should thus increase the probability for the former more than for the latter. Note that despite the ratio $\frac{40}{50}$ being equal to $\frac{4}{5}$, the magnitude of the updated probability should not be the same because the size of the respective batches are different.

Another limitation of the *Weak Estimator Batch* algorithm is that it must be run often because each estimation update is only by a small number. This could potentially cause unnecessary overhead. Thus it might be a good idea to improve the estimation such that it doesn't need to be run as often.

6.3 Future Work

- Future work might look at existing heuristic algorithms regarding the Asymmetric Traveling Salesman Problem (TSP with precedence constraints) and their application for firewall rule optimisation. The reason being that a TSP with precedence constraints is the same problem as that of optimising a firewall with dependency relationships rule order.
- Perform experiments using more real world policies and network traffic data.
- Perform experiments on larger firewall policies.
- Regarding the problem discussed in section 6.2.1, revisiting the experiments done in this thesis using a more controlled testing

environment, where we can guarantee no packet loss between virtual machine instances.

- Update the firewall optimiser created in this thesis to be run in parallel, by having the rule re-ordering algorithm be one thread, the weak estimator algorithm be another thread, and finally have a thread that ensures communication between the two algorithms.
- A new type of firewall optimisation. Create a program able to read the fingerprint of a network (i.e the current state of the traffic), generate an optimised firewall based on the fingerprint and then store them in a database. Finally, the traffic reading program will be able to read the current traffic and apply the appropriate firewall based on the fingerprint from the database.

Chapter 7

Conclusion

The main goal of this thesis was to investigate how to optimise a firewall's rule ordering using network traffic statistics.

The problem statement is addressed by developing two algorithms in order to achieve optimisation of a firewall's rules in a dynamic network. The first algorithm is a rule re-ordering algorithm. It is based on Fulp's simple and naive algorithm for optimising rule re-ordering.

Our algorithm uses more complex metrics for determining rule re-ordering and through experiments it has been shown to reduce the average matching time by as much as 68% more than Fulp's.

The second algorithm is a traffic aware algorithm. It is based on the weak estimator algorithm by Oommen and Rueda. However, it has been modified to accommodate batch updating of rule probabilities rather than having to rely on keeping track of every packet in the system in order to update rule probabilities.

Although this is still only a work in progress, through various experiments, it has been show that the firewall performance optimiser works and is able to re-order rules by using information from the network.

Bibliography

[1] Dave Evans. 'The internet of things: How the next evolution of the internet is changing everything'. In: *CISCO white paper* 1 (2011).

[2] Qi Duan and E. Al-Shaer. 'Traffic-aware dynamic firewall policy management: techniques and applications'. In: *Communications Magazine, IEEE* 51.7 (July 2013), pp. 73–79. ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6553681.

[3] S. Acharya et al. 'Traffic-Aware Firewall Optimization Strategies'. In: *Communications, 2006. ICC '06. IEEE International Conference on*. Vol. 5. June 2006, pp. 2225–2230. DOI: 10.1109/ICC.2006.255101.

[4] Errin W Fulp. 'Optimization of network firewall policies using ordered sets and directed acyclical graphs'. In: *Proc. of IEEE Internet Management Conference*. 2005.

[5] Carsten Benecke. 'A parallel packet screen for high speed networks'. In: *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*. IEEE. 1999, pp. 67–74.

[6] Olivier Paul, Maryline Laurent and Sylvain Gombault. 'A full bandwidth ATM Firewall'. In: *Computer Security-ESORICS 2000*. Springer, 2000, pp. 206–221.

[7] David Newman. *Benchmarking Terminology for Firewall Performance (RFC 2647)*. RFC 2647. http://www.rfc-editor.org/rfc/rfc2647.txt. RFC Editor, Aug. 1999. URL: http://www.rfc-editor.org/rfc/rfc2647.txt.

[8] Karen Scarfone and Paul Hoffman. 'Guidelines on firewalls and firewall policy'. In: *NIST Special Publication* 800 (2009), p. 41.

[9] Elizabeth D Zwicky, Simon Cooper and D Brent Chapman. *Building internet firewalls*. " O'Reilly Media, Inc.", 2000.

[10] Robert Loren Ziegler and Carl B Constantine. *Linux firewalls*. Sams Publishing, 2002.

[11] Jon Postel. *Internet Protocol (RFC 791)*. STD 5. http://www.rfc-editor.org/rfc/rfc791.txt. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc791.txt.

[12] A. Tapdiya and E.W. Fulp. 'Towards Optimal Firewall Rule Ordering Utilizing Directed Acyclical Graphs'. In: *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*. Aug. 2009, pp. 1–6. DOI: 10.1109/ICCCN.2009.5235232.

[13] John R Vacca. *Computer and information security handbook*. Newnes, 2012.

[14] E.S. Al-Shaer and H.H. Hamed. 'Firewall Policy Advisor for anomaly discovery and rule editing'. In: *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*. Mar. 2003, pp. 17–30. DOI: 10.1109/INM.2003.1194157.

[15] Ehab S Al-Shaer and Hazem H Hamed. 'Modeling and management of firewall policies'. In: *Network and Service Management, IEEE Transactions on* 1.1 (2004), pp. 2–10.

[16] Yi Zhang, Yong Zhang and Weinong Wang. 'Optimization of Firewall Filtering Rules by a Thorough Rewriting.' In: *LANOMS*. 2005, pp. 77–88.

[17] Vic Grout and John McGinn. 'Optimisation of Policy-Based Internet Routing using Access Control Lists'. In: *Proceedings of the 9th IFIP/IEEE Symposium on Integrated Network Management*. 2005.

[18] Louis-Philippe Bigras, Michel Gamache and Gilles Savard. 'The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times'. In: *Discrete Optimization* 5.4 (2008), pp. 685–699.

[19] Alexander Schrijver. 'On the history of combinatorial optimization (till 1960)'. In: *Handbooks in Operations Research and Management Science: Discrete Optimization* 12 (2005), p. 1.

[20] Ronald L Graham et al. 'Optimization and approximation in deterministic sequencing and scheduling: a survey'. In: *Annals of discrete mathematics* 5 (1979), pp. 287–326.

[21] B John Oommen and Luis Rueda. 'Stochastic learning-based weak estimation of multinomial random variables and its applications to pattern recognition in non-stationary environments'. In: *Pattern Recognition* 39.3 (2006), pp. 328–341.

[22] Kumpati S Narendra and Mandayam AL Thathachar. *Learning automata: an introduction*. Courier Corporation, 2012.

[23] Mandayam AL Thathachar and Pidaparty S Sastry. *Networks of learning automata: Techniques for online stochastic optimization*. Springer Science & Business Media, 2003.

[24] Luis Rueda and B John Oommen. 'Stochastic automata-based estimators for adaptively compressing files with nonstationary distributions'. In: *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 36.5 (2006), pp. 1196–1200.

[25] B John Oommen and Sudip Misra. 'Fault-tolerant routing in adversarial mobile ad hoc networks: an efficient route estimation scheme for non-stationary environments'. In: *Telecommunication Systems* 44.1-2 (2010), pp. 159–169.

[26] Aleksander Stensby, B John Oommen and Ole-Christoffer Granmo. 'Language detection and tracking in multilingual documents using weak estimators'. In: *Structural, Syntactic, and Statistical Pattern Recognition*. Springer, 2010, pp. 600–609.

[27] Salvatore Sanfilippo. *Hping3*. 2006. (Visited on 10/03/2015).

[28] Vic Grout, John Davies and John McGinn. 'An argument for simple embedded ACL optimisation'. In: *Computer Communications* 30.2 (2007), pp. 280–287.

Appendices

Firewalls

```
#! /bin/bash

if [ "$(id -u)" != "0" ]; then
   echo "This script must be run by root"
   exit 1
fi

# Open everything and then flush all iptables rules
iptables --policy INPUT ACCEPT
iptables --policy OUTPUT ACCEPT
iptables --policy FORWARD ACCEPT
iptables -F

#--------------POLICY-----------------------------------------------------------------------------------
iptables --policy INPUT DROP
iptables --policy OUTPUT DROP
iptables --policy FORWARD DROP
#-------------------------------------------------------------------------------------------------------

#--------------ESTABLISHED/RELATED_TRAFFIC--------------------------------------------------------------
# Allow all INPUT/OUTPUT/FORWARD with the state ESTABLISHED or RELATED
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
#-------------------------------------------------------------------------------------------------------

#--------------DNS_RULES--------------------------------------------------------------------------------
# Allow outgoing dns connections from gateway
iptables -A OUTPUT -p udp --dport 53 -m state --state NEW -j ACCEPT
#-------------------------------------------------------------------------------------------------------

#--------------SSH_RULES--------------------------------------------------------------------------------
# allow incoming ssh connections to gateway from the internet and desktop
# from the internet
iptables -A INPUT -i eth0 -p tcp --dport 22 -m state --state NEW -j ACCEPT

# some forward ssh connections comming from the outside to Machine2
iptables -A FORWARD -p tcp --dport 22 -d 192.168.128.206 -m state --state
    NEW -j ACCEPT
#-------------------------------------------------------------------------------------------------------

#--------------BEGIN_RULES------------------------------------------------------------------------------
iptables -A FORWARD -i eth0 -p udp -s 190.0.0.0/8 --dport 90 -m state
    --state NEW -j ACCEPT -m comment --comment "A"
iptables -A FORWARD -i eth0 -p udp -s 190.1.1.0/24 --dport 90:94 -m state
    --state NEW -j DROP -m comment --comment "B"
iptables -A FORWARD -i eth0 -p udp -s 190.1.2.0/24 -m state --state NEW
    -j DROP -m comment --comment "C"
```

```
45 iptables -A FORWARD -i eth0 -p udp -s 190.1.1.2 --dport 94 -m state
       --state NEW -j ACCEPT -m comment --comment "D"
46
47 iptables -A FORWARD -i eth0 -p tcp -s 190.0.0.0/8 --dport 90 -m state
       --state NEW -j ACCEPT -m comment --comment "E"
48 iptables -A FORWARD -i eth0 -p tcp -s 190.1.1.0/255.255.255.0 --dport 88
       -m state --state NEW -j DROP -m comment --comment "F"
49 iptables -A FORWARD -i eth0 -p tcp -s 190.1.1.2/24 --dport 88:94 -m state
       --state NEW -j ACCEPT -m comment --comment "G"
50 iptables -A FORWARD -i eth0 -p tcp -s 190.1.2.0/24 -m state --state NEW
       -j ACCEPT -m comment --comment "H"
51
52 iptables -A FORWARD -i eth0 -p tcp -d 161.120.33.41 --dport 25 -m state
       --state NEW -j ACCEPT -m comment --comment "I"
53 iptables -A FORWARD -i eth0 -p tcp -s 140.192.37.30 --dport 21 -m state
       --state NEW -j DROP -m comment --comment "J"
54 iptables -A FORWARD -i eth0 -p tcp -d 161.120.33.0/24 --dport 21 -m state
       --state NEW -j DROP -m comment --comment "K"
55 iptables -A FORWARD -i eth0 -p tcp -s 140.192.37.0/24 --dport 21 -m state
       --state NEW -j ACCEPT -m comment --comment "L"
56 iptables -A FORWARD -i eth0 -p tcp -d 161.120.33.0/24 --dport 22 -m state
       --state NEW -j ACCEPT -m comment --comment "M"
57 iptables -A FORWARD -i eth0 -p tcp -s 140.192.37.0/24 --dport 80 -m state
       --state NEW -j DROP -m comment --comment "N"
58 iptables -A FORWARD -i eth0 -p tcp -d 161.120.33.40 --dport 80 -m state
       --state NEW -j ACCEPT -m comment --comment "O"
59 iptables -A FORWARD -i eth0 -p tcp -d 161.120.33.43 --dport 53 -m state
       --state NEW -j ACCEPT -m comment --comment "P"
60 iptables -A FORWARD -i eth0 -p udp -d 161.120.33.43 --dport 53 -m state
       --state NEW -j ACCEPT -m comment --comment "Q"
61 #-----------------------------------------------------------------------------------------------
62
63 iptables -xnvL
```

Listing 7.1: "Original IPtables firewal file"

```
1  #! /bin/bash
2
3  if [ "$(id -u)" != "0" ]; then
4     echo "This script must be run by root"
5     exit 1
6  fi
7
8  # Open everything and then flush all iptables rules
9  iptables --policy INPUT ACCEPT
10 iptables --policy OUTPUT ACCEPT
11 iptables --policy FORWARD ACCEPT
12 iptables -F
13
14 #---------------POLICY-----------------------------------------------------------------------
15 iptables --policy INPUT DROP
16 iptables --policy OUTPUT DROP
17 iptables --policy FORWARD DROP
18 #-----------------------------------------------------------------------------------------------
19
20 #---------------ESTABLISHED/RELATED_TRAFFIC--------------------------------------------------
21 # Allow all INPUT/OUTPUT/FORWARD with the state ESTABLISHED or RELATED
22 iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
23 iptables -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
24 iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
25 #-----------------------------------------------------------------------------------------------
26
27 #---------------DNS_RULES--------------------------------------------------------------------
28 # Allow outgoing dns connections from gateway
29 iptables -A OUTPUT -p udp --dport 53 -m state --state NEW -j ACCEPT
30 #-----------------------------------------------------------------------------------------------
31
```

```
#--------------SSH_RULES--------------------------------------------------------------------------------
# allow incoming ssh connections to gateway from the internet and desktop
# from the internet
iptables -A INPUT -i eth0 -p tcp --dport 22 -m state --state NEW -j ACCEPT

# some forward ssh connections comming from the outside to Machine2
iptables -A FORWARD -p tcp --dport 22 -d 192.168.128.206 -m state --state
    NEW -j ACCEPT
#-------------------------------------------------------------------------------------------------------

#--------------BEGIN_RULES------------------------------------------------------------------------------
iptables -A FORWARD -i eth0 -p udp -s 190.0.0.0/8 --dport 90 -m state
    --state NEW -j ACCEPT -m comment --comment "A"
iptables -A FORWARD -i eth0 -p udp -s 190.1.1.0/24 --dport 90:94 -m state
    --state NEW -j DROP -m comment --comment "B"
iptables -A FORWARD -i eth0 -p udp -s 190.1.2.0/24 -m state --state NEW
    -j DROP -m comment --comment "C"
iptables -A FORWARD -i eth0 -p udp -s 190.1.1.2 --dport 99 -m state
    --state NEW -j ACCEPT -m comment --comment "D"

iptables -A FORWARD -i eth0 -p tcp -s 190.0.0.0/8 --dport 90 -m state
    --state NEW -j ACCEPT -m comment --comment "E"
iptables -A FORWARD -i eth0 -p tcp -s 190.1.1.0/255.255.255.0 --dport 88
    -m state --state NEW -j DROP -m comment --comment "F"
iptables -A FORWARD -i eth0 -p tcp -s 190.1.1.2/24 --dport 88:94 -m state
    --state NEW -j ACCEPT -m comment --comment "G"
iptables -A FORWARD -i eth0 -p tcp -s 190.1.2.0/24 -m state --state NEW
    -j ACCEPT -m comment --comment "H"
#-------------------------------------------------------------------------------------------------------

iptables -xnvL
```

Listing 7.2: "Small version of IPtables firewal file"

Firewall Script for Opening It Up Again

```
# Open everything and then flush all iptables rules
iptables --policy INPUT ACCEPT
iptables --policy OUTPUT ACCEPT
iptables --policy FORWARD ACCEPT
iptables -F
```

Listing 7.3: "Script for removing firewall"

Traffic Generator and Input Files

This is the program used to test the firewall optimisation setup.

```
#! /usr/bin/env python

import os
import sys
import argparse
import logging
import csv
import subprocess
import random
import pprint
```

```
11
12 def read_zipf(filename):
13    with open(filename, 'rb') as f:
14       lines = [line.rstrip('\n') for line in f]
15    return lines
16
17 def read_file(cmd, filename):
18    hping3_cmds = []
19    skip = True
20    with open(filename, 'rb') as f:
21       for row in csv.reader(f):
22          if skip:
23             skip = False
24             continue
25
26          proto, src_ip, src_port, dst_ip, dst_port =
                ",".join(row).split(',')
27          new_cmd = cmd
28
29          if proto == "udp":
30             new_cmd += " --udp"
31
32          if src_ip == "*":
33             new_cmd += " --rand-source"
34          else:
35             new_cmd += " --spoof " + src_ip
36
37          new_cmd += " -s " + src_port
38
39       new_cmd += " -S" # set SYN FLAG
40
41          if dst_ip == "*":
42             new_cmd += " --rand-dest x.x.x.x"
43          else:
44             new_cmd += " " + dst_ip
45
46          new_cmd += " -p " + dst_port
47
48          hping3_cmds.append(new_cmd)
49    return hping3_cmds
50
51 def weighted_random_choice(choices):
52    total = sum(choices.values())
53    logging.debug(choices.values())
54    logging.debug(type(total))
55    logging.debug("total: %f" % total)
56
57    rand_val = random.uniform(0, total)
58    logging.debug("rand_val: %f" % rand_val)
59    current = 0
60    count = 0
61    for key, value in choices.items():
62       logging.debug("####################ITERATION
             %d#######################" % count)
63       current += value
64       logging.debug("current: %f" % current)
65       logging.debug("value: %f" % value)
66       logging.debug("key: %s" % key)
67       if current > rand_val:
68          logging.debug("The returned key: %s" % key)
69          return key
70       count += 1
71
72 def generate_zipf_distribution(N, s):
73    zipf_dist = []
74    sum_z = 0
75    for i in range(1, N + 1):
76       sum_z += 1.0/i**s
```

```
77
78      for i in range(1, N + 1):
79          k = i
80          tmp = (1.0/k**s)/sum_z
81          zipf_dist.append(tmp)
82
83      logging.debug("the generated distribution:")
84      logging.debug(zipf_dist)
85      sum_dist = sum(zipf_dist)
86      logging.debug("the sum of the distribution is: %f" % sum_dist)
87
88      pprint.pprint(zipf_dist)
89      random.shuffle(zipf_dist)
90      pprint.pprint(zipf_dist)
91      exit(0)
92      logging.debug("distribution after shuffle:")
93      logging.debug(zipf_dist)
94      return zipf_dist
95
96  # send traffic from a random source address to a destination address
97  # -n: numeric mode, wont do a lookup for IP
98  # -V: verbose mode
99  # --flood: send packets as fast as possible without waiting for reply
100 # --udp: udp mode, default mode is tcp
101 # --spoof <hostname>: spoof source address
102 # --rand-source: random source address
103 # --rand-dest: random destination address based on user specified rules
104 # i.e: 10.0.0.x
105 # --interval [u]<number>: wait time before sending a pacet in
        [micro]seconds
106 # --fast: alias for -i u10000 (10 packets/s)
107 # --faster: alias for -i u1 (faster than fast)
108 # --interface: choose an interface
109 # --count <number>: how many packets to send
110 # -D: or --debug
111 # sudo hping3 -VDn --faster --count 10 --rand-source 127.0.0.1
112 # sudo hping3 -VDn -I lo --faster --count 10 --rand-source --rand-dest
        127.0.0.x
113 #cmd = "hping3 -VDn --faster --count 10 --rand-source 127.0.0.1"
114 def hping3(cmd):
115     print "\n" + cmd
116     command = cmd.split()
117     try:
118         #with open(os.devnull, 'w') as devnull:
119         # subprocess.check_call(command, stdout=devnull) # we will get 100%
              packet loss because we spoof src IPs, so replys won't go
              anywhere
120         # we will get 100% packet loss because we spoof src IPs, so replys
              won't go anywhere
121         p = subprocess.Popen(command, stdout=subprocess.PIPE)
122         p.stdout.read(1)
123         p.communicate()
124     except (subprocess.CalledProcessError, KeyboardInterrupt) as e:
125         repr(e)
126
127 def main():
128     parser = argparse.ArgumentParser(description="A program that generates
          custom network traffic using hping3",
          formatter_class=argparse.RawTextHelpFormatter)
129
130     parser.add_argument("-v", "--verbose", help="increase output
          verbosity", action="store_true")
131     parser.add_argument("-D", "--debug", help="debug for hping3",
          action="store_true")
132     parser.add_argument("-n", "--numeric", help="numeric output only for
          hping3, no ip lookups", action="store_true")
133     parser.add_argument("-I", "--interface", help="the interface hping3
          should use", type=str, required=True)
```

```
134     #parser.add_argument("-c", "--count", help="number of packet hping3
            should send", type=str, required=False)
135     parser.add_argument("-c", "--count", help="total number of packet to
            send", type=str, required=True)
136     parser.add_argument("--flood", help="tell hping3 to send pacekts as
            fast as possible without waiting for replies", action="store_true")
137     parser.add_argument("-f", "--file", help="file with a list of source
            and destination addresses and ports", type=str, required=True)
138     parser.add_argument("-z", "--zipf", help="file containing a zipf
            distribution", type=str, required=False)
139     args = parser.parse_args()
140
141     cmd = "hping3 "
142     #count = False
143     total_count = 0
144
145     if args.verbose:
146        logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
               %(pathname)s - p%(process)s - %(levelname)s -
               %(funcName)s:%(lineno)d - %(message)s')
147        cmd += " -V"
148
149     logging.debug("Program Start.")
150
151     if args.debug:
152        cmd += " -D"
153
154     if args.numeric:
155        cmd += " -n"
156
157     if args.interface:
158        cmd += " -I " + args.interface
159
160     if args.count:
161        #cmd += " -c " + args.count
162        #count = True
163        cmd += " -c 1"
164        total_count = int(args.count)
165     #else:
166     # cmd += " -c 1"
167
168     if args.flood:
169        cmd += " --flood"
170     else:
171        cmd += " --faster"
172
173     if args.file:
174        cmd_lst = read_file(cmd, args.file) # a list of cmds in the same
               order as in args.file
175
176     zipf_dist = []
177     if args.zipf:
178        zipf_dist = read_zipf(args.zipf)
179     else:
180        zipf_dist = generate_zipf_distribution(len(cmd_lst), 1.2)
181
182     choice_dict = {}
183     for i in range(len(cmd_lst)):
184        choice_dict[cmd_lst[i]] = float(zipf_dist[i])
185
186    # pprint.pprint(cmd_lst)
187    # pprint.pprint(zipf_dist)
188    # pprint.pprint(choice_dict)
189
190     logging.debug(choice_dict)
191     count = 0
192     try:
193        while total_count > 0:
```

```
194             print "Total Count: " + str(total_count)
195             print "Count: " + str(count)
196             if count == 1000:
197                zipf_shuffle = random.shuffle(zipf_dist)
198                choice_dict = {}
199                for i in range(len(cmd_lst)):
200                   choice_dict[cmd_lst[i]] = float(zipf_dist[i])
201                pprint.pprint(choice_dict)
202             cmd = weighted_random_choice(choice_dict)
203             try:
204                hping3(cmd)
205             except KeyboardInterrupt:
206                print "exiting hping3 command function"
207             total_count -= 1
208             count += 1
209       except KeyboardInterrupt:
210          print "Exiting the program"
211          exit(0)
212
213   if __name__ == '__main__':
214       main()
```

Listing 7.4: "The final traffic generator program"

```
1   proto,src_ip,src_port,dst_ip,dst_port
2   udp,190.0.0.2,100,*,90
3   udp,190.1.1.10,50,*,92
4   udp,190.1.2.20,55,*,80
5   udp,190.1.1.2,60,*,94
6   tcp,190.1.20.3,85,*,90
7   tcp,190.1.1.8,39,*,88
8   tcp,190.1.1.2,20,*,89
9   tcp,190.1.2.90,51,*,190
10  tcp,*,11,161.120.33.41,25
11  tcp,140.192.37.30,9,*,21
12  tcp,*,130,161.120.33.40,21
13  tcp,140.192.37.20,90,*,21
14  tcp,*,2001,161.120.33.101,22
15  tcp,140.192.37.111,2333,*,80
16  tcp,*,1,161.120.33.40,80
17  tcp,*,2,161.120.33.43,53
18  udp,*,3,161.120.33.43,53
```

Listing 7.5: "Original information file"

```
1   proto,src_ip,src_port,dst_ip,dst_port
2   udp,190.0.0.2,100,*,90
3   udp,190.1.1.10,50,*,92
4   udp,190.1.2.20,55,*,80
5   udp,190.1.1.2,60,*,99
6   tcp,190.1.20.3,85,*,90
7   tcp,190.1.1.8,39,*,88
8   tcp,190.1.1.2,20,*,89
9   tcp,190.1.2.90,51,*,190
```

Listing 7.6: "Small version of original file"

Rule Ordering Program and Input Files

The program takes two files as input, a dependency relationship file and a firewall file (which are given above)

```
#! /usr/bin/env python
import sys
import time
import datetime
import argparse
import logging
import csv
import re
import operator
import pprint

class Rule:
   def __init__(self, name, rule, prob, pos, preceding, succeeding):
      self.name = name # Unique name based on the hash of the rule
      self.rule = rule # The actual rule
      self.prob = prob # The matching probability for this rule
      self.pos = pos # Position in firewall
      self.preceding = preceding # A list of preceding rules representing
          the precedence relationships
                             # between this rule and the rules in the list
      self.succeeding = succeeding # A list of succeeding rules
          representing the reverse precedence
                             # relationships between this rule and the
                                 rules in the list
   def get_average_matching_time_estimate(self):
      return float(self.prob * self.pos)

   def get_average_matching_time_true(self, prob):
      return float(self.pos * prob)

def get_id(rule):
   pattern = "--comment \"([\w+])\""
   regexp_obj = re.search(pattern, rule)
   if regexp_obj:
      return regexp_obj.group(1)
   else:
      logging.debug("No match found, this should not happen")
      exit(1)

def read_iptables(filename):
   iptables_preamble = []
   iptables_rules = []
   begin_rules = False

   with open(filename, 'r') as f:
      for line in f:
         if "BEGIN_RULES" in line:
            begin_rules = True

         if line.startswith("\n") or line.startswith('#'): # skip newlines
             and comments
            continue

         if begin_rules:
            iptables_rules.append(line.strip())
         else:
            iptables_preamble.append(line.strip())
   return iptables_preamble, iptables_rules

def read_DAG(filename):
   rules = {} # empty dictionary of rules in the DAG
   rules_list = [] # empty list of hashed rules in the original order
   count = 1
   skip = True
   with open(filename, 'rb') as f:
      for row in csv.reader(f):
         if skip:
```

```
64              skip = False
65              continue
66           name = row[0]
67           prob = float(row[1])
68           pos = count
69           if row[2]:
70              preceding = row[2:]
71           else:
72              preceding = []
73           rule = " " # place holder
74           rules[name] = Rule(name, rule, prob, pos, preceding,
                 succeeding=[])
75           rules_list.append(name)
76           count += 1
77     return rules, rules_list
78
79  def write_firewall(preamble, body):
80     logging.debug(preamble)
81     logging.debug(body)
82     ts = time.time()
83     st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d_%H:%M:%S')
84     filename = st + "_fw.rc"
85     with open(filename, 'wb') as f:
86        for rule in preamble:
87           f.write(rule + "\n")
88
89        f.write("\n")
90
91        for rule in body:
92          f.write(rule + "\n")
93     return filename
94
95  def set_succeeding(rules):
96     for rule in rules:
97        rule_name = rules[rule].name
98        if rules[rule].preceding:
99           for r in rules[rule].preceding:
100             rules[r].succeeding.append(rule_name)
101
102 def find_max(rules, rule, lst):
103    logging.debug("rule: %s" % rule)
104    logging.debug("lst: ")
105    logging.debug(lst)
106    if lst == []:
107       #return rules[rule].pos
108       #return len(rules) + 1
109       return -1
110    max_pos = 0
111    for r in lst:
112       r_pos = rules[r].pos
113       if max_pos < r_pos:
114          max_pos = r_pos
115    return max_pos
116
117 def find_min(rules, rule, lst):
118    logging.debug("rule: %s" % rule)
119    logging.debug("lst: ")
120    logging.debug(lst)
121    if lst == []:
122       #return rules[rule].pos
123       #return -1
124       return len(rules) + 1
125    min_pos = len(rules) + 1
126    for r in lst:
127       r_pos = rules[r].pos
128       if min_pos > r_pos:
129          min_pos = r_pos
130    return min_pos
```

```
131
132 def swap(rules, rx, ry):
133    logging.debug("Swapping rule %s and %s" % (rx, ry))
134    temp_pos = rules[rx].pos
135    rules[rx].pos = rules[ry].pos
136    rules[ry].pos = temp_pos
137
138 # we just want to get the ones with highest probability as far up as
       possible
139 # to find the optimal placement of all rules would not be practical in
       regards to time
140 # since it would take a very long time to do an exhaustive search to find
       the optimal solution
141 # we use a heuristic solution that simply tries to get as many of the
       rules with higer prbability
142 # as high up as possible
143 def swapping(rules):
144    for rx in rules:
145       logging.debug("RX: %s" % rx)
146       delta_max = 0
147       delta_max_rule = None
148       for ry in rules:
149          logging.debug("RY: %s" % ry)
150          delta_new = 0
151          if rx != ry:
152             logging.debug("RX:%s != RY:%s" % (rx, ry))
153             logging.debug("RX.pos: %d, RY.pos: %d" % (rules[rx].pos,
                   rules[ry].pos))
154             if rules[rx].pos < rules[ry].pos: # rx is higher up in the
                   firewall rules list
155                logging.debug("rx.pos < ry.pos: RX is higher up than RY")
156                rx_preceding_min_pos = find_min(rules, rx,
                      rules[rx].preceding)
157                ry_succeeding_max_pos = find_max(rules, ry,
                      rules[ry].succeeding)
158                logging.debug("RX Preceding Min Pos: %d" %
                      rx_preceding_min_pos)
159                logging.debug("RY Succeeding Max Pos: %d" %
                      ry_succeeding_max_pos)
160                #if rules[rx].pos > ry_succeeding_min_pos and rules[ry].pos
                      < rx_preceding_max_pos:
161                if rules[rx].pos > ry_succeeding_max_pos and rules[ry].pos
                      < rx_preceding_min_pos:
162                   if rules[rx].prob < rules[ry].prob:
163                      logging.debug("rules[rx], rx=%s: " % rx)
164                      logging.debug(rules[rx])
165                      logging.debug("rules[ry], ry=%s: " % ry)
166                      logging.debug(rules[ry])
167                      delta_new = (rules[ry].prob - rules[rx].prob) *
                            (rules[ry].pos - rules[rx].pos)
168                      logging.debug("delta_new: %d" % delta_new)
169                      if delta_max < delta_new:
170                         delta_max = delta_new
171                         delta_max_rule = ry
172                         logging.debug("delta_max(=%d) < delta_new, ry: %s"
                               % (delta_max, ry))
173                   else:
174                      logging.debug("ry has lower prob than rx")
175                else:
176                   logging.debug("RX.pos is greater than
                         ry_succeeding_min_pos and RY.pos is less than
                         rx_preceding_max_pos")
177             else: # ry is higher up in the firewall rules list
178                logging.debug("ry.pos < rx.pos: RY is higher up than RX")
179                ry_preceding_min_pos = find_min(rules, ry,
                      rules[ry].preceding)
180                rx_succeeding_max_pos = find_max(rules, rx,
                      rules[rx].succeeding)
```

```
181                 logging.debug("RY Preceding Min Pos: %d" %
                        ry_preceding_min_pos)
182                 logging.debug("RX Succeeding Max Pos: %d" %
                        rx_succeeding_max_pos)
183                 #if rules[ry].pos > rx_succeeding_min_pos and rules[rx].pos
                        < ry_preceding_max_pos:
184                 if rules[ry].pos > rx_succeeding_max_pos and rules[rx].pos
                        < ry_preceding_min_pos:
185                    if rules[ry].prob < rules[rx].prob:
186                       logging.debug("rules[ry], ry=%s: " % ry)
187                       logging.debug(rules[ry])
188                       logging.debug("rules[rx], rx=%s: " % rx)
189                       logging.debug(rules[rx])
190                       delta_new = (rules[rx].prob - rules[ry].prob) *
                              (rules[rx].pos - rules[ry].pos)
191                       if delta_max < delta_new:
192                          delta_max = delta_new
193                          delta_max_rule = ry
194                          logging.debug("delta_max(=%d) < delta_new, ry: %s"
                                 % (delta_max, ry))
195                    else:
196                       logging.debug("rx has lower prob than ry")
197                 else:
198                    logging.debug("RY.pos is greater than
                         rx_succeeding_min_pos and RX.pos is less than
                         ry_preceding_max_pos")
199          if delta_max > 0:
200             print "swapping: " + rules[rx].rule + " and " +
                    rules[delta_max_rule].rule
201             swap(rules, rx, delta_max_rule)
202          else:
203             logging.debug("delta_max = %d" % delta_max)
204             continue
205       return delta_max
206
207    def print_rules(rules):
208       for r in rules:
209          attrs = vars(rules[r])
210          logging.debug(', '.join("%s: %s" % item for item in attrs.items()))
211
212       for r in (sorted(rules.values(), key=operator.attrgetter('pos'))):
213          pprint.pprint(vars(r))
214
215    def init(filename_dep, filename_fw):
216       logging.debug("Program Start.")
217
218       filename = filename_dep
219       iptables_filename = filename_fw
220
221       rules, rules_list = read_DAG(filename)
222       iptables_preamble, iptables_rules = read_iptables(iptables_filename)
223       iptables_rules.pop() # remove the last element in the list
224
225       amte = 0 # average match time estimate
226       for rule in iptables_rules:
227          rid = get_id(rule)
228          rules[rid].rule = rule # set the rules field in the relevant object
229          amte += rules[rid].get_average_matching_time_estimate()
230
231       set_succeeding(rules)
232
233       print("The average Match Time Estimate for the initial firewall is:
              %f" % amte)
234       return rules, iptables_preamble
235
236       #length = len(rules_list)
237       #for i in range(1):
238       # swapping(rules)
```

```
239
240 def main():
241     parser = argparse.ArgumentParser(description="A firewall optimisation
            algorithm utilising huristics and swapping")
242     parser.add_argument("-v", "--verbose", help="increase output
            verbosity", action="store_true")
243     parser.add_argument("-f", "--filename", help="the firewall rules file
            with precedece relationships", type=str, required=True)
244     parser.add_argument("-i", "--iptables-file", help="the iptables bash
            script", type=str, required=True)
245     args = parser.parse_args()
246
247     if args.verbose:
248         #logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
                %(levelname)s - %(message)s')
249     logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
            %(pathname)s - p%(process)s - %(levelname)s -
            %(funcName)s:%(lineno)d - %(message)s')
250
251     logging.debug("Program Start.")
252
253     filename = args.filename
254     iptables_filename = args.iptables_file
255
256     rules, rules_list = read_DAG(filename)
257     iptables_preamble, iptables_rules = read_iptables(iptables_filename)
258     iptables_rules.pop() # remove the last element in the list
259
260     for rule in iptables_rules:
261         rid = get_id(rule)
262         rules[rid].rule = rule # set the rules field in the relevant object
263
264     set_succeeding(rules)
265
266     #for r in (sorted(rules.values(), key=operator.attrgetter('pos'))):
267     # pprint.pprint(vars(r))
268     #exit(0)
269
270     length = len(rules_list)
271     for i in range(1):
272         swapping(rules)
273
274     print "\nAfter swapping algorithm"
275     for r in rules:
276         attrs = vars(rules[r])
277         logging.debug(', '.join("%s: %s" % item for item in attrs.items()))
278
279     for r in (sorted(rules.values(), key=operator.attrgetter('pos'))):
280         pprint.pprint(vars(r))
281
282 if __name__ == '__main__':
283     main()
```

Listing 7.7: "The final rule ordering program"

```
1  name,probability,precedence_relationships
2  A,0.05882352941,B,C
3  B,0.05882352941,D
4  C,0.05882352941,
5  D,0.05882352941,
6  E,0.05882352941,G,H
7  F,0.05882352941,G
8  G,0.05882352941,
9  H,0.05882352941,
10 I,0.05882352941,
11 J,0.05882352941,L
12 K,0.05882352941,L
```

```
L,0.05882352941,
M,0.05882352941,
N,0.05882352941,O
O,0.05882352941,
P,0.05882352941,
Q,0.05882352941,
```

Listing 7.8: "Original Relationship file"

```
name,probability,precedence_relationships
A,0.125,B,C
B,0.125,D
C,0.125,
D,0.125,
E,0.125,G,H
F,0.125,G
G,0.125,
H,0.125,
```

Listing 7.9: "Small version of Relationship file"

Traffic Aware Program and Firewall Optimiser

This program contains both the weak estimator algorithm and code for combining the rule ordering algorithm with the weak estimator algorithm to create the optimised dynamic firewall setup.

```
#! /usr/bin/env python
import sys
import time
import datetime
import subprocess
import argparse
import logging
import re
import csv
import operator
import pprint

import rule_ordering

#glob_pkt_cnt_thrsh = 1000 # global packet count threshold
#glob_pkt_cnt_cur = 0 # current global packet count
glob_pkt_cnt_tot = 0 # total global packet count

#def write_avg_mtch_tme(tru_avg, est_avg, base_avg, filename):
# global glob_pkt_cnt_tot
# string = str(glob_pkt_cnt_tot) + "," + str(tru_avg) + "," +
    str(est_avg) + "," + str(base_avg)
# print string
# #ts = time.time()
# #st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d_%H:%M:%S')
# #filename = "average_matching_time_" + st + ".csv"
# with open(filename, 'a') as f:
# f.write(string)
# f.write("\n")

def write_avg_mtch_tme(est_avg, filename):
   global glob_pkt_cnt_tot
   string = str(glob_pkt_cnt_tot) + "," + str(est_avg)
```

```
33      print string
34      with open(filename, 'a') as f:
35          f.write(string)
36          f.write("\n")
37
38  def find_trend(wndw):
39      wndw_len = len(wndw)
40      # find the average avg_mtch_tme of all elements except the latest
41      avg_avg_mtch_tme = sum(wndw[0:(wndw_len - 1)])/(wndw_len - 1)
42      # if the avg_avg_mtch_tme is 10% grater than the latest value
43      if (avg_avg_mtch_tme * 1.1) > wndw[-1]:
44          return True
45      return False
46  #def calc_average_match_time(rules, rules_obj, rules_zipf, static_fw,
        filename):
47  # global glob_pkt_cnt_tot
48  # tru_avg_mtch_tme = 0
49  # est_avg_mtch_tme = 0
50  # base_avg_mtch_tme = 0
51  #
52  # lst = []
53  #
54  # # find tru_avg_mtch_tme for the whole firewall
55  # for r in rules_zipf:
56  # tru_avg_mtch_tme +=
        rules_obj[r].get_average_matching_time_true(rules_zipf[r])
57  #
58  # # find est_avg_mtch_tme for the whole firewall
59  # for r in rules:
60  # est_avg_mtch_tme += rules_obj[r].get_average_matching_time_estimate()
61  #
62  # for r in static_fw:
63  # base_avg_mtch_tme += rules_zipf[r[0]] * r[1]
64  #
65  # print tru_avg_mtch_tme
66  # print est_avg_mtch_tme
67  # print base_avg_mtch_tme
68  # write_avg_mtch_tme(tru_avg_mtch_tme, est_avg_mtch_tme,
        base_avg_mtch_tme, filename)
69
70  def calc_average_match_time(rules, rules_obj, filename):
71      est_avg_mtch_tme = 0
72
73      # find est_avg_mtch_tme for the whole firewall
74      for r in rules:
75          est_avg_mtch_tme +=
                rules_obj[r].get_average_matching_time_estimate()
76
77      print est_avg_mtch_tme
78      write_avg_mtch_tme(est_avg_mtch_tme, filename)
79      return est_avg_mtch_tme
80
81  def read_DAG(filename):
82      rules = {} # empty dictionary of rules in the DAG
83      skip = True
84      with open(filename, 'rb') as f:
85          for row in csv.reader(f):
86              if skip:
87                  skip = False
88                  continue
89              name = row[0]
90              prob = float(row[1])
91              pkts = 0 # initial value
92              lst = [prob, pkts]
93              rules[name] = lst
94      return rules
95
96  def get_pkts(rule):
```

```
97     pattern = "^\s*([\d]+).* /\* ([\w]+) \*/"
98     regexp_obj = re.search(pattern, rule)
99     if regexp_obj:
100       return regexp_obj.group(1), regexp_obj.group(2)
101    else:
102       logging.debug("No match found for the pattern:")
103       logging.debug(pattern)
104       logging.debug("in line:")
105       logging.debug(rule)
106       return None, None
107
108 def read_iptables_stats():
109    process = subprocess.Popen(['iptables', '-xnvL'],
           stdout=subprocess.PIPE)
110    out, err = process.communicate()
111    out = out.splitlines()
112    pkts_per_rule = {}
113    for line in out:
114       pkts, name = get_pkts(line)
115       if pkts and name:
116          pkts_per_rule[name] = int(pkts)
117    return pkts_per_rule
118
119 def rule_update(preamble, rules_obj, rules):
120    body = []
121    # simply overwrite the value list with an updated value list
122    for r in rules:
123       logging.debug("rules_obj[%s].prob: %f" % (r, rules_obj[r].prob))
124       logging.debug("rules[%s][0]: %f" % (r, rules[r][0]))
125       rules_obj[r].prob = rules[r][0]
126       # REMEBER THAT EACH TIME A NEW REORDER IS NEEDED ALL THE PACKET
              STATS IN IPTABLES ARE RESET
127       # THIS MUST BE DONE IN THE CODE AS WELL.
128       tmp_lst = rules[r]
129       tmp_lst.pop()
130       tmp_lst.append(0)
131       rules[r] = tmp_lst
132
133    for i in range(10):
134       logging.debug("################### SWAPPING ITERATION: %d
              ############################" % i)
135       delta_max = rule_ordering.swapping(rules_obj)
136       if delta_max <= 0: # there was no swap that happend in the last
              iteration, we can return early
137          break
138
139    for r in (sorted(rules_obj.values(), key=operator.attrgetter('pos'))):
140       #pprint.pprint(vars(r))
141       body.append(r.rule)
142
143    filename = rule_ordering.write_firewall(preamble, body)
144
145    process = subprocess.Popen(['bash', filename], stdout=subprocess.PIPE)
146    out, err = process.communicate()
147
148    if err:
149       print err
150       exit(0)
151
152 def weak_estimator(rules, lmbda):
153    #global glob_pkt_cnt_cur
154    global glob_pkt_cnt_tot
155
156    my_sum = 0.0
157    for i in rules:
158          my_sum = my_sum + rules[i][0]
159    logging.debug("sum of rules: %f" % my_sum)
160
```

```
161     # previous sample minus current sample to find how many packets have
            been matched
162     # in total since the prevous sample.
163     pkts_per_rule = read_iptables_stats()
164
165     # calculate total amount of packet matches for last sample
166     M_old = 0
167     for v in rules.values():
168        M_tmp = v[1]
169        M_old += M_tmp
170     # calculate total amount of packet matches for current sample
171     M_new = sum(pkts_per_rule.values())
172     # calculate total amount of packet matches since the last sample
173     M = M_new - M_old
174     #glob_pkt_cnt_cur += M
175     glob_pkt_cnt_tot += M
176
177     if M > 0:
178        for rule in rules:
179           cur_prob = rules[rule][0]
180           cur_pkts = rules[rule][1]
181           upd_pkts = pkts_per_rule[rule]
182           M_i = float(upd_pkts - cur_pkts) # amount of packet matches since
                  last sample for one rule
183           upd_prob = cur_prob + (lmbda * (M_i/M - cur_prob))
184           logging.debug("Rule: %s\nprev_prob: %f\nprev_pkts: %d\ncur_pkts:
                  %d\nM_i: %d\n cur_prob: %f" %
185              (rule, cur_prob, cur_pkts, upd_pkts, M_i, upd_prob))
186           rules[rule] = [upd_prob, upd_pkts]
187     else:
188        logging.debug("nothing has happend since last sample, no need to do
              anything")
189
190  def main():
191     #global glob_pkt_cnt_thrsh
192     #global glob_pkt_cnt_cur
193     global glob_pkt_cnt_tot
194
195     parser = argparse.ArgumentParser(description="A firewall optimisation
           algorithm utilising huristics and swapping")
196     parser.add_argument("-v", "--verbose", help="increase output
           verbosity", action="store_true")
197     parser.add_argument("-f", "--filename", help="the firewall rules file
           with precedece relationships", type=str, required=True)
198     parser.add_argument("-fw", "--firewall", help="the iptables firewall
           rules file", type=str, required=True)
199     parser.add_argument("-ws", "--window-size", help="window size of
           rule", type=int, required=True)
200     args = parser.parse_args()
201
202     if args.verbose:
203        #logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
              %(levelname)s - %(message)s')
204        logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -
              %(pathname)s - p%(process)s - %(levelname)s -
              %(funcName)s:%(lineno)d - %(message)s')
205
206     logging.debug("Program Start.")
207
208     filename = args.filename
209     filename_fw = args.firewall
210
211     # initialise the rules dictionaries
212     rules_obj, iptables_preamble = rule_ordering.init(filename,
           filename_fw) # dictionary of rules objects
213     logging.debug(rules_obj)
214     rules = read_DAG(filename)
215     initial_pkts_per_rule = read_iptables_stats()
```

```
    lmbda = 0.2
    #weak_estimator(rules, lmbda)
    for i in initial_pkts_per_rule:
       cur_pkts = initial_pkts_per_rule[i]
       tmp_lst = rules[i]
       tmp_lst.pop()
       tmp_lst.append(0)
       rules[i] = tmp_lst

    ###################################EXPERIMENT
        CODE#################################################
    # both zipf distributions are as long as the rules dict as long as the
        smaller firewalls are used
    #zipf_dist_X = [0.062132360042377745, 0.049922963174044155,
        0.04149198453973711, 0.03534873454458779,
    # 0.42862930043528075, 0.18657173946957867, 0.11469285134920415,
        0.08121006644518972]

    #zipf_dist_Y = [0.42862930043528075, 0.18657173946957867,
        0.11469285134920415, 0.08121006644518972,
    # 0.062132360042377745, 0.049922963174044155, 0.04149198453973711,
        0.03534873454458779]

    #zipf_dist_X = [0.05231293211266767, 0.04203311417119375,
        0.034934571436956136, 0.029762203612392586,
    # 0.3608885205692724, 0.1570858524343425, 0.09656673820770563,
        0.06837558866128435,
    # 0.025839378080981655, 0.022770526259177373, 0.020309622245726824,
        0.018295975609411506,
    # 0.016620383676689998, 0.015206155421475426, 0.013997921608126592,
        0.012954751559851152,
    # 0.012045764332744381]

    #zipf_dist_Y = [0.1570858524343425, 0.029762203612392586,
        0.05231293211266767, 0.018295975609411506,
    # 0.06837558866128435, 0.012954751559851152, 0.034934571436956136,
        0.020309622245726824,
    # 0.012045764332744381, 0.025839378080981655, 0.09656673820770563,
        0.3608885205692724,
    # 0.022770526259177373, 0.016620383676689998, 0.013997921608126592,
        0.04203311417119375,
    # 0.015206155421475426]

    #count = 0
    #rules_zipf_X = {}
    #rules_zipf_Y = {}
    #static_fw = []
    #for r in (sorted(rules_obj.values(), key=operator.attrgetter('pos'))):
    # static_tuple = (r.name, r.pos)
    # static_fw.append(static_tuple)
    # rules_zipf_X[r.name] = zipf_dist_X[count]
    # rules_zipf_Y[r.name] = zipf_dist_Y[count]
    # count += 1
    #pprint.pprint(rules_zipf_X)
    #pprint.pprint(rules_zipf_Y)
    #pprint.pprint(static_fw)
    #last_swap_avg_mtch_tme = 0
    avg_mtch_tme_wndw = []
    max_wndw_size = args.window_size
    ts = time.time()
    st = datetime.datetime.fromtimestamp(ts).strftime('%Y-%m-%d_%H:%M:%S')
    log_filename = "performance_triggered_experiment_wndw_size_" +
        str(max_wndw_size) + "_" + st + ".csv"
    ###################################EXPERIMENT
        CODE#################################################

    #last_swap_avg_mtch_tme = calc_average_match_time(rules, rules_obj,
        log_filename)
```

```
267     try:
268        while True:
269           weak_estimator(rules, lmbda)
270           logging.debug("After estimation")
271           logging.debug(rules)
272           #rule_update(iptables_preamble, rules_obj, rules)
273           #print("\ncur packet count: %d\ntotal packet count: %d" %
                  (glob_pkt_cnt_cur, glob_pkt_cnt_tot))
274           #if glob_pkt_cnt_cur >= glob_pkt_cnt_thrsh:
275           tmp_avg_mtch_tme = calc_average_match_time(rules, rules_obj,
                  log_filename)
276           length = len(avg_mtch_tme_wndw)
277           if length < max_wndw_size:
278              avg_mtch_tme_wndw.append(tmp_avg_mtch_tme)
279           else:
280              avg_mtch_tme_wndw.pop(0) # fifo list, first in first out, we
                     only want the lates values
281              avg_mtch_tme_wndw.append(tmp_avg_mtch_tme)
282
283           if length == max_wndw_size and find_trend(avg_mtch_tme_wndw): #
                  return true if trend is increasing, else false
284              rule_update(iptables_preamble, rules_obj, rules)
285            # glob_pkt_cnt_cur = 0
286
287           #if glob_pkt_cnt_tot >= 10000:
288           # calc_average_match_time(rules, rules_obj, rules_zipf_Y,
                  static_fw, log_filename)
289           #else:
290           # calc_average_match_time(rules, rules_obj, rules_zipf_X,
                  static_fw, log_filename)
291
292           #if glob_pkt_cnt_tot >= 20000:
293           # print "20000 packets matched, experiment over"
294           # exit(0)
295           time.sleep(2)
296     except KeyboardInterrupt:
297        print "exiting the program"
298        exit(0)
299
300  if __name__ == '__main__':
301     main()
```

Listing 7.10: "The final traffic aware program"